

Lecture Notes in Computer Science

1551

Gopal Gupta (Ed.)

Practical Aspects of Declarative Languages

First International Workshop, PADL'99
San Antonio, Texas, USA, January 1999
Proceedings



Springer

Springer

Berlin

Heidelberg

New York

Barcelona

Hong Kong

London

Milan

Paris

Singapore

Tokyo

Gopal Gupta (Ed.)

Practical Aspects of Declarative Languages

First International Workshop, PADL'99
San Antonio, Texas, USA, January 18-19, 1999
Proceedings



Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editor

Gopal Gupta
New Mexico State University, Department of Computer Science
New Science Hall (Box 30001, MS CS)
Stewart Street, Las Cruces, NM 88003, USA
E-mail: gupta@cs.nmsu.edu

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Practical aspects of declarative languages : first international workshop ;
proceedings / PADL '99, San Antonio, Texas, USA, January 18 - 19, 1999. Gopal
Gupta (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London
; Milan ; Paris ; Singapore ; Tokyo : Springer, 1998
(Lecture notes in computer science ; 1551)
ISBN 3-540-65527-1

CR Subject Classification (1998): D.3, D.1, F.3, D.2, I.2.3

ISSN 0302-9743

ISBN 3-540-65527-1 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

© Springer-Verlag Berlin Heidelberg 1999
Printed in Germany

Typesetting: Camera-ready by author
SPIN 10693172 06/3142 - 5 4 3 2 1 0 Printed on acid-free paper

Preface

Declarative languages have traditionally been regarded by the mainstream computing community as too impractical to be put to practical use. At the same time, traditional conferences devoted to declarative languages do not have issues related to practice as their central focus. Thus, there are few forums devoted to discussion of practical aspects and implications of newly discovered results and techniques related to declarative languages. The goal of the First International Workshop on Practical Aspects of Declarative Languages (PADL) is to bring together researchers, practitioners and implementors of declarative languages to discuss practical issues and practical implications of their research results. The workshop was held in San Antonio, Texas, during January 18-19, 1999. This volume contains its proceedings.

Fifty three papers were submitted in response to the call for papers. These papers were written by authors belonging to twenty one countries from six continents. Each paper was assigned to at least two referees for reviewing. Twenty four papers were finally selected for presentation at the workshop. Many good papers could not be included due to the limited duration of the workshop. The workshop included invited talks by Mark Hayden of DEC/Compaq Systems Research Center, speaking on “Experiences Building Distributed Systems in ML,” and Mark Wallace of Imperial College Center for Planning And Resource Control (IC-PARC), speaking on “ECLiPSe: Declarative Specification and Scalable Implementation.” The workshop also included a panel discussion on “What can declarative paradigms contribute to meet the challenges of Software Engineering?”

The workshop is sponsored and organized by COMPULOG AMERICAS (<http://www.cs.nmsu.edu/~complog>), a network of research groups dedicated to promoting research in logic programming and related areas, by the Association for Logic Programming (<http://www.cwi.nl/projects/alp>), and the Department of Computer Science, New Mexico State University (<http://www.cs.nmsu.edu>). The workshop is being held in cooperation with the ACM Special Interest Group on Programming Languages (ACM SIGPLAN). The support of many individual was crucial to the success of this workshop. My thanks to Enrico Pontelli for his tremendous help in managing and organizing every aspect of the workshop. He also took upon himself the time-consuming task of developing and maintaining the PADL’99 web-pages. Thanks are also due to Krzysztof Apt, President of ALP, and David S. Warren, past President of ALP, for their support, help, and encouragement, to Andrew Appel, POPL General Chair, and Donna Baglio and Maria Triver of ACM, for answering various organizational questions, and to the referees for their prompt reviewing of papers. My thanks to the program committee members for all their help in reviewing and their advice. Finally, my thanks to all the authors who took interest in PADL’99 and submitted papers.

Program Committee

M. Carlsson	SICS, Sweden
T. Chikayama	University of Tokyo, Japan
B. Demoen	KU Leuven, Belgium
H. Gill	DARPA, USA
G. Gupta	New Mexico State University, USA (chair)
M. Hermenegildo	Technical University of Madrid, Spain
B. Jayaraman	SUNY Buffalo, USA
N. Jones	University of Copenhagen, Denmark
R. Kieburtz	Oregon Graduate Institute, USA
M. Martelli	University of Genoa, Italy
S. Peyton Jones	Microsoft Research, UK
E. Pontelli	New Mexico State University, USA
I.V. Ramakrishnan	SUNY Stonybrook, USA
D. Schmidt	Kansas State University, USA
V. Santos Costa	University of Porto, Portugal
P. van Hentenryck	Brown University, USA
D.S. Warren	SUNY Stonybrook, USA
P. Wadler	Lucent Technologies Bell Labs, USA

Organizing Committee

E. Pontelli	New Mexico State University, USA (chair)
G. Gupta	New Mexico State University, USA

List of Referees

A. K. Bansal, C. Baoqiu, M. Bruynooghe, F. Bueno, M. Carlsson, M. Carro, M. M. T. Chakravarty, H. Davulcu, B. Demoen, I. Dutra, M. Florido, G. Gupta, M. Huth, J. Hatcliff, N. Jacobs, B. Jayaraman, N. D. Jones, S. Peyton Jones, R. B. Kieburtz, K. Narayan Kumar, J. P. Leal, S. W. Loke, R. Lopes, W. Luk, J. Marino, R. F. P. Marques, M. Martelli, S. McKeever, T. Mogensen, A. H. Nieva, O. Olsson, E. Pontelli, G. Puebla, C. R. Ramakrishnan, P. Rao, R. Rocha, A. Roychoudhury, V. Santos Costa, D. Schmidt, M. H. B. Sorensen, K. Sagonas, S. Singh, A. Stoughton, P. Szeredi, A. Tiwari, P. van Hentenryck, P. Wadler, and D. S. Warren.

Table of Contents

Software Engineering

Automated Benchmarking of Functional Data Structures.....	1
<i>G.E. Moss and C. Runciman</i>	
NP-SPEC: An Executable Specification Language for Solving All Problems in NP	16
<i>M. Cadoli, L. Palopoli, A. Schaerf, and D. Vasile</i>	
Prototyping a Requirements Specification through an Automatically Generated Concurrent Logic Program	31
<i>P. Letelier, P. Sánchez, and I. Ramos</i>	
Multi-agent Systems Development as a Software Engineering Enterprise.....	46
<i>M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, and F. Zini</i>	

Innovative Applications I

From Functional Animation to Sprite-Based Display	61
<i>C. Elliott</i>	
Beyond Pretty-Printing: Galley Concepts in Document Formatting Combinators.....	76
<i>W. Kahl</i>	
Lambda in Motion: Controlling Robots with Haskell.....	91
<i>J. Peterson, P. Hudak, and C. Elliott</i>	

Implementation

CHAT: The Copy-Hybrid Approach to Tabling	106
<i>B. Demoen and K. Sagonas</i>	
The Influence of Parallel Computer Architectural Parameters on the Performance of Parallel Logic Programming Systems	122
<i>M.G. Silva, I.C. Dutra, R. Bianchini, and V. Santos Costa</i>	
Or-Parallelism within Tabling	137
<i>R. Rocha, F. Silva, and V. Santos Costa</i>	

Declarative Query Processing

Mnesia – A Distributed Robust DBMS for Telecommunications Applications.....	152
<i>H. Mattsson, H. Nilsson, and C. Wikström</i>	
An AQUA-Based Intermediate Language for Evaluating an Active Deductive Object-Oriented Language.....	164
<i>B. Siddabathuni, S.W. Dietrich, and S.D. Urban</i>	
Implementing a Declarative String Query Language with String Restructuring	179
<i>R. Hakli, M. Nykänen, H. Tamm, and E. Ukkonen</i>	

Systems Applications

Client-Side Web Scripting with HaskellScript	196
<i>E. Meijer, D. Leijen, and J. Hook</i>	
MCORBA: A CORBA Binding for Mercury	211
<i>D. Jeffery, T. Dowd, and Z. Somogyi</i>	

Analysis

Dead Code Elimination through Dependent Types	228
<i>H. Xi</i>	
Multiple Specialization of WAM Code	243
<i>M. Ferreira and L. Damas</i>	
A Flexible Framework for Dynamic and Static Slicing of Logic Programs	259
<i>W.W. Vasconcelos</i>	

Innovative Applications II

Applying Logic Programming to Derive Novel Functional Information of Genomes	275
<i>A.K. Bansal and P. Bork</i>	
An Application of Action Theory to the Space Shuttle	290
<i>R. Watson</i>	
Developing a Declarative Rule Language for Applications in Product Configuration	305
<i>T. Soininen and I. Niemelä</i>	

Constraint Programming

University Timetabling Using Constraint Logic Programming	320
<i>H-J. Goltz and D. Matzke</i>	
Constraint-Based Resource Allocation and Scheduling in Steel Manufacturing	335
<i>M. Carlsson, P. Kreuger, and E. Åström</i>	
Using Constraints in Local Proofs for CLP Debugging	350
<i>C. Lai</i>	

Declarative Languages and Software Engineering (Invited)

A Return to Elegance: The Reapplication of Declarative Notation to Software Design	360
<i>D.A. Schmidt</i>	
ECLiPSe : Declarative Specification and Scaleable Implementation	365
<i>M. Wallace and J. Schimpf</i>	
Author Index	367

Automated Benchmarking of Functional Data Structures

Graeme E. Moss and Colin Runciman

Department of Computer Science, University of York, UK
`{gem,colin}@cs.york.ac.uk`

Abstract. Despite a lot of recent interest in purely functional data structures, for example [Ada93, Oka95, BO96, Oka96, OB97, Erw97], few have been benchmarked. Of these, even fewer have their performance qualified by how they are used. But how a data structure is used can significantly affect performance. This paper makes three original contributions. (1) We present an algorithm for generating a benchmark according to a given use of data structure. (2) We compare use of an automated tool based on this algorithm, with the traditional technique of hand-picked benchmarks, by benchmarking six implementations of random-access list using both methods. (3) We use the results of this benchmarking to present a decision tree for the choice of random-access list implementation, according to how the list will be used.

1 Motivation

Recent years have seen renewed interest in purely functional data structures: sets [Ada93], random-access lists [Oka95], priority queues [BO96], arrays [OB97], graphs [Erw97], and so on. But, empirical performance receives little attention, and is usually based on a few hand-picked benchmarks. Furthermore, the performance of a data structure usually varies according to how it is used, yet this is mostly overlooked.

For example, Okasaki [Oka95] uses five simple benchmarks to measure the performance of different implementations of a list providing random access. He points out that three of the benchmarks use random access, and two do not. However, all the benchmarks are single-threaded. How do the data structures perform under non-single-threaded use? We simply do not know.

Okasaki presents many new data structures in his thesis [Oka96], but without measurements of practical performance. He writes in a section on future work: “The theory and practice of benchmarking [functional] data structures is still in its infancy.”

How can we make benchmarking easier and more reliable? A major problem is finding a range of benchmarks that we know use the data structure in different ways. If we could generate a benchmark according to a well-defined use of the data structure, we could easily make a table listing performance against a range of uses.

To make precise “the use of a data structure” we need a model. Section 2 defines such a model: a *datatype usage graph*, or DUG. Section 2 also defines a *profile*, summarising the important characteristics of a DUG. Section 3 gives an algorithm for generating a benchmark from a profile. Section 4 introduces a benchmarking kit, called *Auburn*, that automates benchmarking using the algorithm of Sections 3. Section 4 then compares benchmarking six implementations of random-access lists manually against using Auburn. Section 5 discusses related work. Section 6 concludes and discusses future work.

Some of the details of this paper are only relevant to the language we use: Haskell, a pure functional language using lazy evaluation. Such details are clearly indicated.

2 Modelling Datatype Usage

How can we capture the way an application uses a data structure? Take the *Sum* benchmark of [Oka95] as an example of an application. Sum uses an implementation of random-access lists (see Fig. 1) to build a list of n integers using *cons*, and then sum this list using *head* and *tail*. Code for Sum is given in Fig. 2(a).

Let us use a graph to capture how Sum uses the list operations. Let a node represent the result of applying an operation, and let the incoming arcs indicate the arguments taken from the results of other operations. Let any other arguments be included in the label of the node. Figure 2(b) shows this graph.

Node 1 represents the result of *empty*, which is an empty list. Node 2 represents the result of applying *cons* to 1 and *empty*, which is a list containing just the integer 1. And so on, till node $n + 1$ represents a list of n copies of the integer 1. This is how Sum builds a list of n integers.

Node $n + 2$ represents the result of applying *head* to this list, which is the first element in the list. Node $n + 3$ represents the result of applying *tail* to this list, which is all but the first element. Node $n + 4$ represents the result of applying *head* to the list of node $n + 3$, giving the second element. Every other element of the list is removed in the same way, till node $3n$ represents the last element, and node $3n + 1$ represents the empty list. This is how Sum sums the list of n integers.

The authors introduced such a graph in [MR97], given the name *datatype usage graph*, or DUG. The definition was informal in [MR97] but we shall now give a brief formal definition. To abstract over many competing data structures providing similar operations, we insist on a DUG describing the use of an ADT. The same DUG can then describe the use of any implementation of that ADT. We restrict an ADT to being *simple*.

Definition 1 (Simple ADT)

A *simple* ADT provides a type constructor T of arity one, and only operations over types shown in Table 1.

```

empty  :: List a
cons   :: a → List a → List a
tail   :: List a → List a
update :: List a → Int → a → List a
head   :: List a → a
lookup :: List a → Int → a

```

Fig. 1. The signature of a random-access list abstract datatype (ADT).

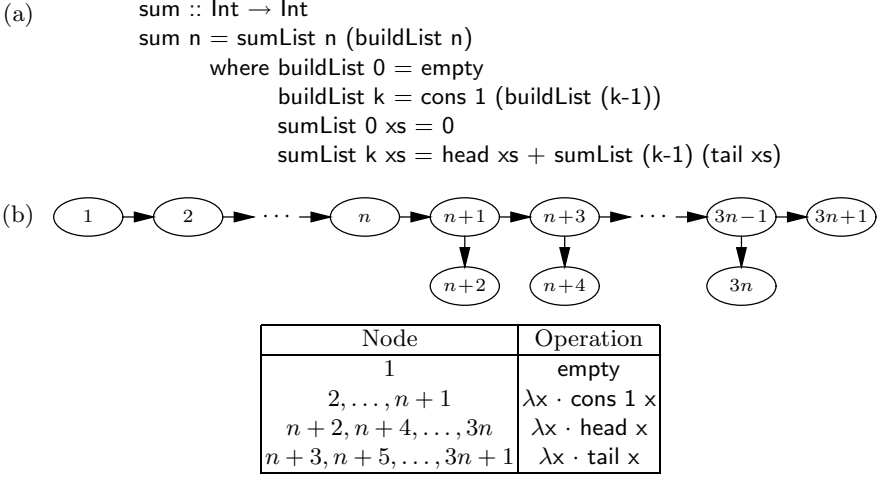


Fig. 2. The Sum benchmark of [Oka95] using the operations of the ADT in Fig. 1. (a) Code. (b) DUG.

Table 1. The types of operations allowed in a simple ADT providing a type constructor T , where a is a type variable. Any value of type $T \ a$ is called a *version*. Each ADT operation is given a single *role*. A *generator* takes no versions, but returns a version. A *mutator* takes at least one version, and returns a version. An *observer* takes at least one version, but does not return a version.

Argument Types	Result Type	Role
a, Int	$T \ a$	Generator
$T \ a$ (at least one), a, Int	$T \ a$	Mutator
$T \ a$ (at least one), a, Int	$a, \text{Int}, \text{Bool}$	Observer

Example 1

The random-access list ADT, whose signature is given in Fig. 1 is a simple ADT, providing the following: a type constructor `List`; a generator `empty`; mutators `cons`, `tail`, and `update`; and observers `head` and `lookup`.

The nodes are labelled by a function η with partial applications of ADT operations.

Definition 2 (DUG PAP)

A DUG PAP (Partial Application) is a function f obtained by supplying zero or more atomic values as arguments to any operation g of a simple ADT. We say that f is a PAP of g .

Example 2

The DUG PAPs that label nodes of Fig. 2(b) are: `empty`, $\lambda x. \text{cons } 1 \ x$, $\lambda x. \text{head } x$, and $\lambda x. \text{tail } x$.

To indicate which argument corresponds to which arc, if more than one arc is incident to a node, we order the arcs with positive integers using a function τ . To indicate the order of evaluation, we label each node with a positive integer using a function σ .

Definition 3 (DUG)

Given a simple ADT \mathcal{A} , a DUG is a directed graph \mathcal{G} with the following functions:

- A function η labelling every node with a PAP of an operation of \mathcal{A}
- A function τ , which for all nodes with two or more incoming arcs, when restricted to those arcs, is a bijection with the set $\{1, \dots, j\}$ for some $j \geq 2$
- A bijection σ from the nodes to the set $\{1, \dots, n\}$ for some $n \geq 0$

The following conditions must also be true:

C_1 If w is a successor of v , then $\sigma(w) > \sigma(v)$.

C_2 Every node labelled by η as an observer has no arcs from it.

The reasons for imposing these conditions are given in the problems list of Sect. 3.1.

Example 3

A DUG is shown in Fig. 2(b). The function η is given by the table, τ is redundant as no node has more than one incoming arc, and σ is given by the number labelling each node.

To help identify important characteristics of use, we compact important information from a DUG into a *profile*. One piece of important information is the degree of *persistence*, that is, the re-use of a previously mutated data structure.

Definition 4 (Persistent Arc)

Consider the arcs from some v to v_1, \dots, v_k . Let v_i be the mutator node ordered earliest by σ , if it exists. Any arc from v to some v_j ordered *after* v_i represents an operation done *after* the first mutation, and is therefore *persistent*.

The properties making up a profile reflect important characteristics of datatype usage. They are fashioned to make benchmark generation easy. Other reasons for choosing these properties are not discussed here – see [Mos99].

Definition 5 (Profile)

The profile of a DUG comprises five properties:

- *Generation weights*: The ratio of the number of nodes labelled with each generator.
- *Mutation-observation weights*: The ratio of the number of arcs leading to each mutator or observer.
- *Mortality*: The fraction of generator and mutator nodes that have no arcs leading to a mutator.
- *Persistent mutation factor* (PMF): The PMF is the fraction of arcs to mutators that are persistent.
- *Persistent observation factor* (POF): The POF is the fraction of arcs to observers that are persistent.

Example 5

The profile of the DUG of Fig. 2(b) is as follows: the generation weights are trivial as there is only one generator, the mutation-observation weights are

$$\text{cons} : \text{tail} : \text{update} : \text{head} : \text{lookup} = n : n : 0 : n : 0,$$

the mortality is $1/(2n + 1)$, and the PMF and the POF are both zero.

3 Benchmark Generation

If we can generate a DUG with a given profile, then a DUG *evaluator* can act as a benchmark with a well-defined use of data structure (the given profile). There are many DUGs with a given profile, but choosing just one might introduce bias. Therefore we shall use probabilistic means to generate a *family* of DUGs that have *on average* the profile requested.

3.1 DUG Generation

Here are some problems with DUG generation, and the solutions we chose:

- *Avoid ill-formed applications of operations*. For example, we need to avoid forming applications such as `head empty`. We introduce a *guard* for each operation, telling us which applications are well-defined. See the section *Shadows and Guards* below for more details.
- *Order the evaluation*. We cannot reject an ill-formed application of an operation till we know all the arguments. Therefore a node must be constructed after its arguments. Under the privacy imposed by ADTs and the restrictions imposed by lazy evaluation, we can only order the evaluation of the observers. For simplicity, we evaluate the observers as they are constructed. The function σ therefore gives the order of construction of all nodes, and the order of evaluation of observer nodes. This is condition C_1 of Defn. 3.

- *Choose non-version arguments.* Version arguments are of type $\top \mathbf{a}$, and can be chosen from the version results of other nodes. Choosing non-version arguments in the same way is too restrictive – for example, where does the argument of type \mathbf{a} for the first application of `cons` come from? Within the type of each operation, we instantiate \mathbf{a} to `Int`, so we need only choose values of type `Int`. For simplicity, we avoid choosing *any* non-version arguments from the results of other nodes. This is condition C_2 of Defn. 3.

Shadows and Guards. To avoid creating ill-defined applications whilst generating a DUG, we maintain a *shadow* of every version. The shadow contains the information needed to avoid an ill-defined application of any operation to this version. We create the shadows using shadow operations: the type of a shadow operation is the same except that every version is replaced by a shadow. A *guard* takes the shadow of every version argument, and returns the ranges of non-version arguments for which the application is well-defined. Any combination of these non-version arguments must provide a well-defined application.

For example, for random-access lists we maintain the length of the list associated with a version node. This allows us to decide whether we can apply `head` to this node: if the length is non-zero, then we can; otherwise, we cannot. Similarly, applications of other operations can be checked. Figure 3(a) gives shadow operations for random-access lists and Fig. 3(b) gives guards.

The Algorithm. We build a DUG one node at a time. Each node has a *future*, recording which operations we have planned to apply to the node, in order. The first operation in a node’s future is called the *head operation*. The nodes with a non-empty future together make up the *frontier*. We specify a minimum and a maximum size of the frontier. The minimum size is usually 1, though a larger value encourages diversity. Limiting the maximum size of the frontier caps space usage of the algorithm.

Figure 4 shows the algorithm. We make a new node by choosing version arguments from the frontier. We place each argument in a buffer according to the argument’s head operation. When a buffer for an operation f is full (when it contains as many version arguments as f needs), we empty the buffer of its contents vs , and attempt to apply f to vs using the function *try_application*.

Calling *try_application*(f, vs) uses the guard of operation f to see whether f can be applied to version arguments vs . If one of the integer subsets returned by the guard is empty, no choice of integer arguments will make the application well-formed, and so we must abandon this application. Otherwise, the guard returns a list of integer subsets, from which we choose one integer each, to give the integer arguments is . Applying f to vs and is gives a new node.

Planning for the Future. We plan the future of a new node v as follows. We use the five profile properties to make the DUG have the profile requested, on average. Mortality gives the probability that the future of v will contain no mutators. If the future will contain at least one mutator, then the fraction of

(a) **type** Shadow = Int

<code>empty_S</code>	<code>:: Shadow</code>	<code>empty_S</code>	<code>= 0</code>
<code>cons_S</code>	<code>:: Int → Shadow → Shadow</code>	<code>cons_S x s</code>	<code>= s+1</code>
<code>tail_S</code>	<code>:: Shadow → Shadow</code>	<code>tail_S s</code>	<code>= s-1</code>
<code>updates_S</code>	<code>:: Shadow → Int → Int → Shadow</code>	<code>updates_S s i x</code>	<code>= s</code>
(b) <code>empty_G</code>	<code>:: Bool</code>	<code>empty_G</code>	<code>= True</code>
<code>cons_G</code>	<code>:: Shadow → [IntSubset]</code>	<code>cons_G s</code>	<code>= [All]</code>
<code>tail_G</code>	<code>:: Shadow → Bool</code>	<code>tail_G s</code>	<code>= s>0</code>
<code>update_G</code>	<code>:: Shadow → [IntSubset]</code>	<code>update_G s</code>	<code>= [0:...(s-1), All]</code>
<code>head_G</code>	<code>:: Shadow → Bool</code>	<code>head_G s</code>	<code>= s>0</code>
<code>index_G</code>	<code>:: Shadow → [IntSubset]</code>	<code>index_G s</code>	<code>= [0:...(s-1)]</code>

Fig. 3. (a) Shadow operations for random-access lists, maintaining the length of the list. (b) Guards for random-access lists. A guard returns a list of integer subsets, one for each non-version argument, in order. (Haskell does not support functions over tuples of arbitrary size – necessary for the application of an arbitrary guard – so we are forced to use lists.) If an operation does not take any non-version arguments, a boolean is returned. The type `IntSubset` covers subsets of the integers. The value `m:..:n` indicates the subset from `m` to `n` inclusive, and `All` indicates the maximum subset.

persistent mutators should equal PMF. Every application of a mutator bar the first is persistent. Therefore the number m of persistent mutators is given by:

$$\frac{m}{m+1} = \text{PMF} \Rightarrow m = \frac{\text{PMF}}{1 - \text{PMF}}$$

Hence a random variable with mean $\text{PMF}/(1 - \text{PMF})$ gives the number of additional mutators. Note that a PMF of 0 guarantees each node is mutated at most once.

The mutation-observation weights ratio allows us to calculate the average number r of applications of observers per application of a mutator. We assume a mutator made v , and let a random variable with mean r decide the number of observers in the future of v . Typically the number of applications of generators is very small, and so this assumption is reasonable. The number of observers ordered after the first mutator is given by a random variable with mean POF. Finally, we choose which mutators and observers to use from probabilities given by the mutation-observation weights ratio.

Note that condition C_2 of Defn. 3 implies that every observer node has no future.

```

generate_dug() :=
  dug := {}
  frontier := {}
   $\forall f \cdot \text{buffer}(f) := \{\}$ 
  while #dug < final_dug_size do
    if #frontier < min_frontier_size then
      g := choose a generator using generation weights ratio
      try_application(g, {})
    else-if #frontier > max_frontier_size then
      remove a node from frontier
    else
      v := remove a node from frontier
      f := remove head operation of v
      add v to buffer(f)
      if #buffer(f) = number of version arguments taken by f then
        vs := buffer(f)
        buffer(f) := {}
        try_application(f, vs)
      fi
    fi
  od

try_application(f, vs) :=
  int_subsets := apply guard of operation f to shadows of vs
  if each set in int_subsets is not empty then
    is := choose one integer from each set in int_subsets
    v := make node by applying f to version arguments vs and integers is
    shadow of v := apply shadow of f to shadows of vs
    if f is an observer then
      future of v := empty
    else
      plan future of v
    fi
    add v to dug
    if v has a non-empty future then
      add v to frontier
    fi
  fi
  add each node in vs with a non-empty future to frontier

```

Fig. 4. DUG generation algorithm.

3.2 DUG Evaluation

A DUG evaluator uses an implementation of the ADT to evaluate the result of every observation. The nodes are constructed in the same order that they were generated. An observer node is evaluated immediately. This fits the intended behaviour (see the second problem listed for DUG generation).

4 An Example of a Benchmarking Experiment

Auburn is an automated benchmarking kit built around the benchmark generation of Sect. 3. We shall benchmark six implementations of random-access lists (1) using Auburn, and (2) using benchmarks constructed manually. We shall then compare these two methods.

4.1 Aim

We aim to measure the performance of six implementations of random-access lists: *Naive* (ordinary lists), *Threaded Skew Binary* (Myers' stacks [Mye83]), *AVL* (AVL trees [Mye84]), *Braun* (Braun trees [Hoo92]), *Slowdown* (Kaplan and Tarjan's dequeues [KT95]), and *Skew Binary* (Okasaki's lists [Oka95]). We will qualify this performance by two properties of use:

- *Lookup factor*. The number of applications of `lookup` divided by the number of applications of ordinary list operations. We use just two settings: 0 and 1.
- *Update factor*. This is as lookup factor but replacing `lookup` with `update`. Again, we use only two settings: 0 and 1.

There are 4 different combinations of settings of these properties.

4.2 Method

Auburn. We use Auburn version 2.0a. For the latest version of Auburn, see the Auburn Home Page [Aub]. Perform the experiment using Auburn as follows:

- Copy the makefile for automating compilation of benchmarks with the command: `auburnExp`.
- With each random-access list implementation in the current directory, each file ending `List.hs`, make the benchmarks with: `make SIG=List`. This includes making shadows and guards for random-access lists (see Sect. 3.1). Auburn guesses at these from the type signatures of the operations. The makefile will stop to allow the user to check the accuracy of the guess. In the case of random-access lists, it is correct. Restart the makefile with: `make`.
- The makefile also makes a *null implementation*, which implements an ADT in a type-correct but value-incorrect manner. It does a minimum of work. It is used to estimate the overhead in DUG evaluation.

- Make a profile for each of the 4 combinations of properties. Auburn makes a Haskell script to do this. It requires a small change (one line) to reflect the properties and settings we chose.
- Make DUGs from these 4 profiles with: `makeDugs -S 3 -n 100000`. This uses 3 different random seeds for each, creating 12 DUGs, each DUG containing 100000 nodes.
- Run and time each of the 7 DUG evaluators on each of the 12 DUGs. Evaluate each DUG once – internal repetition of evaluation is sometimes useful for increasing the time taken, but we do not need it for this experiment. Take three timed runs of an evaluator to even out any glitches in recorded times. Use: `evalDugs -R 1 -r 3`.
- Process these times with: `processTimes`. This command sums the times for each implementation and profile combination, subtracts the null implementation time, and finally divides by the minimum time over all implementations. This gives an idea of the ratio of work across implementations per profile.

Manual. Perform the experiment without Auburn as follows:

- Construct benchmarks that use random-access lists in manners that cover the 4 properties of use. Take four of the five benchmarks of [Oka95], neglecting Quicksort. Alter them to match one of the 4 properties of use, to force work (as Haskell is lazy), and to construct lists before using them (as it is hard to separate the time of use from the time of construction). Here are the resulting benchmarks:
 - *Sum*. Construct a list of size n using `cons`, and sum every element using `head` and `tail`.
 - *Lookup*. Construct a list of size n using `cons`, and sum every element using `lookup`.
 - *Update*. Construct a list of size n using `cons`, update every element using `update`, update every element twice more, and sum every element using `head` and `tail`.
 - *Histogram*. Construct a list of n zeros using `cons`. Count the occurrence of each integer in an ordinary list of $3n$ pseudo-randomly generated integers over the range $0, \dots, n-1$, using `lookup` and `update` to store these counts. Sum the counts with `head` and `tail` to force the counting.
- Work out what values of n to use in each of the above benchmarks to fix the number of operations done by each benchmark to approximately 100000, for consistency with the Auburn method. Use a loop within the benchmark to repeat the work as necessary.
- Run and time these benchmarks using each implementation (including the null implementation). As with the Auburn method, time three runs, sum these times, subtract the null implementation time, and divide by the minimum time.

Table 2. Ratios of times taken to evaluate benchmarks constructed by Auburn. Null implementation times were subtracted before calculating the ratios. An entry marked “–” indicates the benchmark took too long – the ratio would be larger than for any ratio given for another implementation.

Auburn Results								
Benchmark	Profile Properties		Implementation					
	Lookup Factor	Update Factor	Naive	Thrd. SBin.	AVL	Braun	Slow-down	Skew Binary
DUG Evaluator	0	0	1.0	5.8	127.8	36.3	15.0	18.1
	0	1	1.0	–	8.9	56.5	14.1	3.7
	1	0	–	1.0	1.4	7.2	4.3	3.8
	1	1	–	51.4	1.0	6.9	4.4	3.7

Table 3. As Table 2 but for hand-picked benchmarks.

Manual Results								
Benchmark	Profile Properties		Implementation					
	Lookup Factor	Update Factor	Naive	Thrd. SBin.	AVL	Braun	Slow-down	Skew Binary
Sum	0	0	1.0	2.5	171.6	24.9	18.6	2.8
Update	0	1	–	–	1.0	4.8	3.4	2.6
Lookup	1	0	–	1.0	3.3	9.4	5.8	4.7
Histogram	1	1	–	3.1	1.0	4.7	2.8	3.0

Table 4. As Table 2 but with PMF and POF set to 0.2.

Auburn Results – PMF and POF = 0.2								
Benchmark	Profile Properties		Implementation					
	Lookup Factor	Update Factor	Naive	Thrd. SBin.	AVL	Braun	Slow-down	Skew Binary
DUG Evaluator	0	0	1.0	5.8	37.5	6.9	12.0	13.6
	0	1	1.0	6.2	4.9	10.3	4.7	4.4
	1	0	3.7	1.0	1.9	5.8	3.3	3.1
	1	1	1.7	1.2	1.0	4.0	2.2	2.1

4.3 Results

Tables 2 and 3 give the results. The tables agree on the winner of three of the four combinations of lookup factor and update factor. Naive is the best implementation when no random-access operations are used. Threaded Skew Binary is the best when only lookup operations are used. AVL is the best when both lookup and update operations are used.

But what about when only update operations are used? The Auburn results show Naive as the winner, whereas the manual results show AVL as the winner, with Naive as one of the worst! The probable cause of this difference is that when the relevant DUGs were evaluated, updates on elements towards the rear end of the list were not forced by the only observing operation: `head`. As Naive is very lazy, it benefits greatly. For the manual benchmarks, this does not happen because every element is forced. Adding a maximal sequence of applications of `head` and `tail` to the DUGs and re-timing evaluations produced the same results as the manual benchmarks. This adds weight to our suspicion of the cause of difference between times. As most applications will force most operations, we conclude that AVL is the best implementation when only update operations are used.

Although the Auburn and manual results differ in scale, the order in which they place the implementations are almost always the same. From the results of other experiments, we suspect that the differences are probably due in the main to differences in the sizes of the lists involved. The size of a data structure can significantly affect performance. Unfortunately, this characteristic has proved hard to capture completely. The mutation weights ratio goes some way to capturing size, but neglects the order of applications of mutators. The size of a version can be deduced from the operations used to produce it, but this does not help us to produce versions of a given size: We can measure size, but we cannot completely influence it.

4.4 Comparing Auburn with Manual

Although the description of how to perform this experiment was longer for Auburn than for manual, the user actually has to do less. Auburn automates most of the work, whereas manual benchmarks need to be designed, built, tested, compiled, run, and have the timings collected and analysed. The two most laborious steps for the Auburn user are:

- *Make the shadows and guards.* Auburn can guess at shadows and guards by inspecting the operation types. Auburn manages to guess correctly for many data structures: lists, queues, dequeues, priority queues, and so on, with or without random-access and catenation. For other data structures, minor corrections can be made to this guess. For more exotic data structures, Auburn can generate trivial shadows and guards, from which the user can build their own.

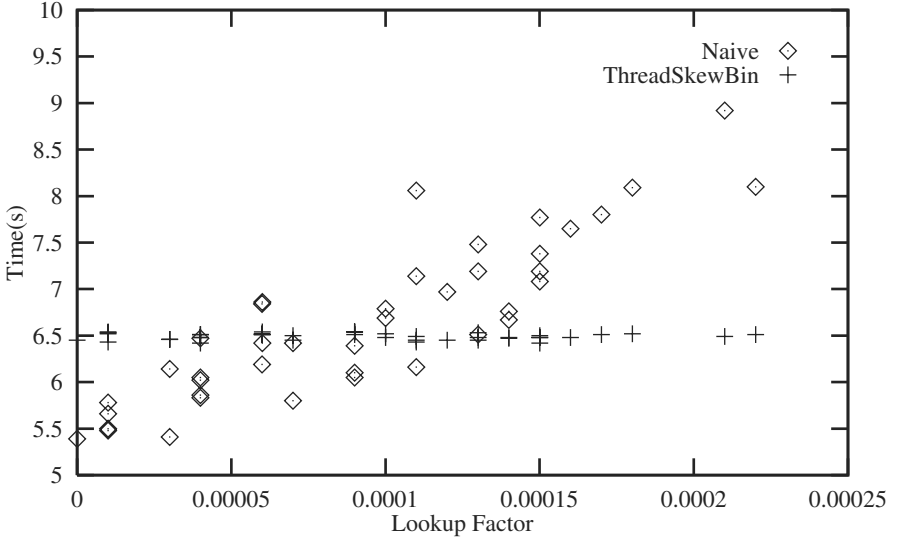


Fig. 5. Times taken to evaluate DUGs with lookup factors calculated from their profiles. Update factor is 0, and each DUG contains 100000 nodes. Each DUG was evaluated three times, and the total time recorded.

-
- *Make the profiles.* Auburn creates a simple script to do this, which needed a small change (one line) to suit our experiment. Further small changes yield other results easily. To illustrate this, let us consider two examples:
 - *Persistence.* How do the results change if we add persistence? For Auburn, we make another small change (one line) to the profiles script, setting PMF and POF to 0.2. We then re-run the experiment with three commands. The results are given in Table 4. Although a few marked changes occur (for example, for Naive with lookup factor non-zero), the winner remains the same in each category. This shows that in this case, persistence does not change which implementation is best. With manual benchmarks, adding persistence would have been a lot harder.
 - *Boundaries.* For update factor equal to 0, we know that Naive is the best for lookup factor equal to 0, and Threaded Skew Binary for lookup factor equal to 1. But where is the boundary at which they perform the same? Again, using Auburn, this requires a small change (one line) to the profiles script, and re-entering three commands. The results are given in Fig. 5. We see that the boundary is approximately 0.0001, which is surprisingly small. With manual benchmarks, the lack of automation would have imposed more work on the user.

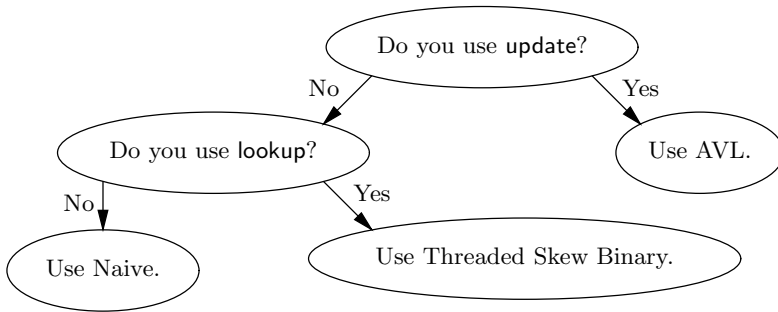


Fig. 6. Decision tree for choosing an implementation of random-access lists.

5 Related Work

The authors published a paper at IFL'97 [MR97] benchmarking queues using Auburn. Auburn was built around queues before being generalised, and this paper shows that Auburn can cope with other data structures. We also extend [MR97] by formalising the definition of a DUG. We also discuss the problems of benchmark generation, and the solutions we chose, including an algorithm for DUG generation. We make a direct comparison of benchmarking using Auburn with benchmarking using the traditional technique of manual benchmarks. We discuss the advantages and disadvantages of Auburn over manual benchmarking, and provide some examples of where Auburn is more useful. Finally we present some advice on which random-access list implementation to use in Haskell, based on the benchmarking results. We know of no other attempt to automate benchmarking of functional data structures.

Okasaki [Oka95] and O'Neill [OB97] benchmark functional random-access lists and arrays, respectively. Okasaki uses five benchmarks, three of which use random-access, and two do not. However, none of the benchmarks use persistence. O'Neill uses two persistent benchmarks, one of which is randomly generated. However, there is no mention of the relative frequency of lookup and update.

6 Conclusions and Future Work

We have formalised a model of how an application uses a data structure. We have described the problems of benchmark generation, the solutions we chose, and a resulting algorithm. We have also used an automated tool (Auburn) based on this algorithm to benchmark six implementations of random-access list. We compared using Auburn to using hand-picked benchmarks, and provided some examples of where Auburn is more useful.

From the results of Sect. 4, we provide a decision tree in Fig. 6 to guide users choosing an implementation of random-access lists. Further experiments using more profiles and more implementations would refine this decision tree.

Future work would involve: lifting the restrictions on ADTs by allowing higher-order operations, and operations between two or more data structures (eg. to and from ordinary lists); lifting the restrictions on DUGs by separating order of construction from order of evaluation, and allowing dependencies on observations; understanding the concept of persistent observations in a lazy language; capturing size of data structure more adequately in the profile; setting down some guidelines on benchmarking.

We dream of a time when a library of functional data structures provides detailed decision trees for *every* ADT implemented.

References

- [Ada93] Stephen R. Adams. Efficient sets – a balancing act. *Journal of Functional Programming*, 3(4):553–561, October 1993.
- [Aub] The Auburn Home Page. <http://www.cs.york.ac.uk/~gem/auburn/>.
- [BO96] Gerth S. Brodal and Chris Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming*, 6(6):839–857, November 1996.
- [Erw97] Martin Erwig. Functional programming with graphs. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 52–65. ACM Press, June 1997.
- [Hoo92] Rob R. Hoogerwoord. A logarithmic implementation of flexible arrays. In *Proceedings of the Second International Conference on the Mathematics of Program Construction*, volume 669 of *LNCIS*, pages 191–207, July 1992.
- [KT95] Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slow-down. In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 93–102, May 1995.
- [Mos99] Graeme E. Moss. *Benchmarking Functional Data Structures*. DPhil thesis, University of York, 1999. To be submitted.
- [MR97] Graeme E. Moss and Colin Runciman. Auburn: A kit for benchmarking functional data structures. In *Proceedings of IFL’97*, volume 1467 of *LNCIS*, pages 141–160, September 1997.
- [Mye83] Eugene W. Myers. An applicative random-access stack. *Information Processing Letters*, 17(5):241–248, December 1983.
- [Mye84] Eugene W. Myers. Efficient applicative data types. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 66–75, 1984.
- [OB97] Melissa E. O’Neill and F. Warren Burton. A new method for functional arrays. *Journal of Functional Programming*, 7(5):487–513, September 1997.
- [Oka95] Chris Okasaki. Purely functional random-access lists. In *Conference Record of FPCA ’95*, pages 86–95. ACM Press, June 1995.
- [Oka96] Chris Okasaki. *Purely Functional Data Structures*. PhD thesis, School of Computer Science, Carnegie Mellon University, September 1996.

NP-SPEC: An Executable Specification Language for Solving All Problems in NP

Marco Cadoli¹, Luigi Palopoli², Andrea Schaerf³, and Domenico Vasile²

¹ Dipartimento di Informatica e Sistemistica
Università di Roma “La Sapienza”, Via Salaria 113, I-00198 Roma, Italy
cadoli@dis.uniroma1.it

<http://www.dis.uniroma1.it/~cadoli>

² Dipartimento di Elettronica Informatica e Sistemistica
Università della Calabria, I-87036 Rende (CS), Italy
[\(palopoli,vasile\)@si.deis.unical.it](mailto:(palopoli,vasile)@si.deis.unical.it)

<http://wwwinfo.deis.unical.it/~palopoli>

³ Dipartimento di Ingegneria Elettrica, Gestionale e Meccanica
Università di Udine, Via delle Scienze 208, I-33100 Udine, Italy
aschaerf@diegm.uniud.it

<http://www.diegm.uniud.it/~aschaerf>

Abstract. In this paper, a logic-based specification language, called NP-SPEC, is presented. The language is obtained extending DATALOG through allowing a limited use of some second-order predicates of predefined form. NP-SPEC programs specify solutions to problems in a very abstract and concise way, and are executable. In the present prototype they are compiled to PROLOG code which is run to construct outputs. Second-order predicates of suitable form allow to limit the size of search spaces in order to obtain reasonably efficient construction of problem solutions. NP-SPEC expressive power is precisely characterized as to express exactly the problems in the class NP.

1 Introduction

In this paper we present the executable specification language NP-SPEC, which allows the user to specify exactly all problems which belong to the complexity class NP.

We start with an example of an NP-SPEC specification, which helps us to highlight the main aspects of the language. The example concerns the famous “3-coloring” problem, which is well-known to be NP-complete (cf. e.g., [GJ79], problem GT4, page 191), and is formally defined as follows.

INSTANCE: Graph $G = (N, E)$

QUESTION: Is G 3-colorable, i.e., does there exist a function
 $col : N \rightarrow \{0, 1, 2\}$ such that $col(u) \neq col(v)$
whenever $\{u, v\} \in E$?

In NP-SPEC, the user can make the following declarations, which specify both an instance (in the **DATABASE** section) and the question (in the **SPECIFICATION** section). In this case, the instance is a graph with six nodes and seven edges, which is 3-colorable.

DATABASE

```
NODE = {1..6};
EDGE = {(1,2), (1,3), (2,3), (6,2), (6,5), (5,4), (3,5)};
```

SPECIFICATION

```
Partition(NODE,coloring,3).
fail <-- edge(X,Y), coloring(X,C), coloring(Y,C).
```

In the current version of NP-SPEC, processing the above specification gives as output a PROLOG program. Running the PROLOG program produces the following output, which represents the colors to be assigned to the nodes so that the input graph is 3-colored.

```
coloring: (1,0), (2,1), (3,2), (4,0), (5,1), (6,0)
```

The underlying language of NP-SPEC is a subset of PROLOG, and more precisely of its function-free fragment DATALOG. In particular:

- the clauses in the **DATABASE** section specify a set of *facts*, i.e., function-free ground atomic first-order formulas;
- a clause in the **SPECIFICATION** section can be a universally quantified function-free first-order definite formula.

The main difference between NP-SPEC and DATALOG is the possibility of using second-order logic in a limited way. Referring to the above example, the clause `Partition(NODE,coloring,3)` has, intuitively, the following meaning: `coloring` is a binary predicate, and its extension can be any set of the following kind:

$$\{\langle n, c \rangle \mid n \text{ is in the extension of } \text{NODE}, c \in \{0, 1, 2\}\}.$$

The intuitive meaning of the **SPECIFICATION** section is: `fail` becomes true if for *all* possible extensions of `coloring`, there are nodes `X` and `Y` such that:

- `X` and `Y` are connected by an edge (cf. atom `edge(X,Y)`), and
- `X` and `Y` are colored with the same color (cf. atoms `coloring(X,C)`, `coloring(Y,C)`).

The execution of the specification is such that as soon as the program finds an extension of `coloring` that does not make `fail` true, it outputs such an extension, or, at the request of the user, all extensions of this kind.

¹ This set has cardinality 3, as specified by the clause.

In general, a specification in NP-SPEC has the following structure:

- The **DATABASE** section is used to specify the instance of a problem. As for the semantics, the extension of the predicates which are declared here (called *extensional* predicates) is always as specified by the set of facts.
- The **SPECIFICATION** section is used to:
 - Declare predicates which are used to represent the search space of the problem (e.g., **coloring**). As for the semantics, the extension of such predicates can be any subset of the Cartesian product (with the appropriate arity) of the active domain. For this reason, we call such predicates *guessed* predicates.
 - Declare constraints binding extensional and guessed predicates, and possibly other predicates (one of them being the predefined predicate **fail**).

The main distinctive features of NP-SPEC are the following:

- It has a completely defined semantics, which is based on an extension of DATALOG known as DATALOG^{CIRC} , defined in [CP98].
- It has a precise connotation in terms of expressive power and computational complexity. In particular, it can specify exactly all problems in the complexity class NP.
- It offers a set of metapredicates, called *tailoring predicates*, which guide the system in the synthesis of algorithms that operate on search spaces of limited size so as to be as efficient as possible.

The language NP-SPEC is meant to be simple to use and to propose intuitive and concise specifications. Our approach is meant to lift the abstraction level at which the problem must be specified for the system to be able to generate automatically the solution code. The user is allowed to take the decisions at such an abstract level, and her/his decisions have an impact in the generated program in a transparent way.

In the rest of the paper, we outline the main technical aspects of NP-SPEC: syntax, semantics, and computational properties. We also briefly address the current prototype and its performances, report more specification examples, and discuss related work.

2 Syntax and Semantics of DATALOG^{CIRC}

As we mentioned, NP-SPEC is based on the language DATALOG^{CIRC} . Actually, the syntax of NP-SPEC extends the one of DATALOG^{CIRC} by means of the tailoring predicates, which are a syntactic extension that makes it easier to specify a problem. For this reason, we first illustrate the syntax and the semantics of DATALOG^{CIRC} . We also briefly address computational properties of DATALOG^{CIRC} , which explain why NP-SPEC is able to specify exactly all problems in NP.

2.1 Syntax of DATALOG^{CIRC}

First of all, we remind the syntax of DATALOG. A DATALOG program [Ull88] is a finite set of universally quantified function-free first-order definite formulae T , i.e., sentences of one of the forms reported below.

$$\begin{aligned} a(\mathbf{X}) &\leftarrow b_1(\mathbf{X}_1), \dots, b_n(\mathbf{X}_n) \\ a(\mathbf{c}) &\leftarrow \end{aligned}$$

where $n \geq 0$, $a(\mathbf{X})$, $b_i(\mathbf{X}_i)$ ($1 \leq i \leq n$), and $a(\mathbf{c})$ are atoms. A clause of the latter kind is called a *fact*.

An atom is *ground* if no variables occur in it. The set of clauses occurring in T is then naturally partitioned in a subset D of facts which constitute the *input database*, and a subset π of non-ground or non-atomic formulae. We assume that no constants appear in π .

A DATALOG^{CIRC} program is a triplet $\langle T; P; Q \rangle$ where T is a DATALOG program and $(P; Q)$ is a partition of the predicates occurring in T . Predicates in P are called *minimized*; predicates in Q are called *guessed*. Intuitively the predicates in Q are those that are guessed by the algorithm, whereas those in P are either calculated or coming from the input database. In fact, the extension of the predicates in the set Q is not determined *a priori*, conversely it is necessary to consider for each of them all possible extensions; that is, all the subsets of the power-set of the Cartesian product of the domains of their attributes. In order to obtain all possible combination among the instances of all predicates of Q it is sufficient to consider their Cartesian product. We call *instance of Q* an element of the latter set. Note that also the predicates belonging to the set P that depend through some clauses upon predicates in Q may have multiple extensions.

The predicates in Q can appear only in the body of the clauses; furthermore, for each predicate p which belongs to P , there must be at least one clause having p in the head which determines its extension, unless p is a database predicate that represents a database input relation D through a set of facts.

The purpose of writing DATALOG^{CIRC} programs is to ask *boolean queries*. Queries are posed by means of *restricted clauses*, which are first-order formulae of the kind

$$(\exists \mathbf{X}) \neg A_1(\mathbf{X}) \vee \dots \vee \neg A_n(\mathbf{X}) \vee e \quad (1)$$

where each A_i ($1 \leq i \leq n$) is a predicate symbol, and e is a 0-ary predicate symbol, i.e., a propositional letter. We note that formula (1) can be equivalently written as

$$((\forall \mathbf{X}) A_1(\mathbf{X}) \wedge \dots \wedge A_n(\mathbf{X})) \rightarrow e$$

2.2 Semantics of DATALOG^{CIRC}

First of all, let us recall the well-known semantics of DATALOG. A (Herbrand) model M of a DATALOG program T is a set of ground atomic formulae such that for each ground instance clause C constructible from T either the head of

C is in M or at least one atom in the body of C does not belong to M . The intersection of all the Herbrand models of a DATALOG program T is itself a model for T [vEK76, Ull88]. This model, denoted LM_T , is called the *least* or *minimum* model of T . The semantics of a DATALOG program relies on its least model.

As for DATALOG^{CIRC} programs, their semantics originates from the non-monotonic formalism of *circumscription* [McC80], and takes into account partition of predicates into $(P; Q)$. Let T be a DATALOG^{CIRC} program. A preorder among the Herbrand models of T is defined as follows.

Definition 1 (*$(P; Q)$ -minimal models*, [Lif85]). *Let M, N be two Herbrand models of a formula T . We write $M \leq_{(P; Q)} N$ if:*

1. *predicates in Q have the same extension in N and M ;*
2. *for each predicate $p \in P$, the extension of p in M is a subset —possibly not proper— of the extension of p in N .*

An Herbrand model M is called $(P; Q)$ -minimal for T if there is no Herbrand model N of T such that $N \leq_{(P; Q)} M$ and $M \not\leq_{(P; Q)} N$. \square

The semantics of a DATALOG^{CIRC} program relies on its Herbrand $(P; Q)$ -minimal models: Let T be a fact-free DATALOG^{CIRC} program, D an input database, and γ a restricted clause, we write $T \wedge D \models_{(P; Q)} \gamma$ if γ is true in all Herbrand $(P; Q)$ -minimal models of $T \wedge D$.

It can be easily seen that DATALOG^{CIRC} is a generalization of DATALOG: A DATALOG program T is just a DATALOG^{CIRC} program $\langle T; P; \emptyset \rangle$ in which the set Q of guessed predicates is empty.

In DATALOG^{CIRC}, we specify the “3-coloring” problem, by means of the fact-free program $\langle T_{co3COL}; P; Q \rangle$ consisting in the following clauses:

$$\begin{aligned} fail &\leftarrow edge(X, Y), red(X), red(Y). \\ fail &\leftarrow edge(X, Y), blue(X), blue(Y). \\ fail &\leftarrow edge(X, Y), green(X), green(Y). \\ has_color(X) &\leftarrow red(X). \\ has_color(X) &\leftarrow blue(X). \\ has_color(X) &\leftarrow green(X). \end{aligned}$$

where $P = \{has_color, fail, edge\}$, $Q = \{red, green, blue\}$. The input database D is constituted by ground atomic instances of $edge$, which is a symmetric extensional predicate encoding the set E of edges of the input graph G in the obvious way. Consider next the query γ defined by the following restricted clause:

$$((\forall X) has_color(X)) \rightarrow fail$$

The intended meaning of the last three rules of T_{co3COL} is to force each node to be colored in some way. The intended meaning of the antecedent of the query γ is that we are interested in $(P; Q)$ -minimal models of $T_{co3COL} \wedge D$ which assign a color to each node. As a matter of fact, we don’t care if a node is assigned more

than one color: If no conflict arises with such an overloading, then we can select in an arbitrary way a color for a node among those which the node is associated with.

As it can be easily verified, $T_{co3COL}, D \not\models_{(P;Q)} \gamma$ if and only if the input graph G is 3-colorable.

2.3 Computational Properties of DATALOG^{CIRC}

As it is evident from the previous example, the problem of determining whether $T \wedge D \models_{(P;Q)} \gamma$ or not, where the input is D , is NP-hard. In [CP98] it is proven that:

- The *data complexity* of DATALOG^{CIRC} , i.e., the input is the database and *not* the fact-free program, is always in NP.
- The *expressiveness* of DATALOG^{CIRC} is such that the language *captures* NP. Roughly, this means that for each problem A in NP, there are a *fixed* fact-free DATALOG^{CIRC} program $\langle T; P; Q \rangle$ and a *fixed* restricted clause γ such that for each instance of d of A encoded as an input database D , it holds that $T \wedge D \models_{(P;Q)} \gamma$ iff d is a “yes” instance of A .
- The *expression complexity* of DATALOG^{CIRC} , i.e., the input is the database *and* the fact-free program, is NE-hard.

This means that the language is capable of specifying exactly all problems which belong to NP. We remind that DATALOG is capable of expressing a strict subset of the polynomial-time problems (as an example, it cannot express the “even” query, which input is a domain C of objects, and which question is: “Is the cardinality of C even?”).

3 The Specification Language NP-SPEC

We now illustrate specifically the language NP-SPEC. In particular, we introduce its basic syntax and the main tailoring predicates currently implemented.

3.1 Syntax of NP-SPEC: Basics

In general, the syntactic differences between DATALOG^{CIRC} and NP-SPEC have been designed with three goals in mind:

1. allowing the user to specify a problem in a more natural way;
2. allowing the user to specify the search space in a way so as to help the interpreter generate an efficient code;
3. not exceeding NP as the data complexity.

The main differences between DATALOG^{CIRC} and NP-SPEC are the following:

- The restricted query clause necessary for specifying a problem always consists in the single literal **fail**.

- It is possible to denote the complement of a predicate, by using the prefix “co_”.
- There are suitable built-in metapredicates for specifying the search space (cf. Subsection 3.2).
- The language is extended by means of arithmetical and relational operators (cf. e.g., programs in Section 4).

In NP-SPEC, in order to identify the predicates belonging to Q and to fix the domains for their attributes, we include in the specification section, for each element of Q , a fact of the following form.

`Subset(<DOMAIN>, <predicate_name>).`

where `Subset` is a built-in metapredicate, and `<DOMAIN>` is a relation that identifies the domain upon which the extension of the predicate is guessed. `<DOMAIN>` is a finite set defined as an input relation, or as an enumeration, or by means of union, intersection, difference, and Cartesian product.

Using the basic syntax, we can specify the question of the “3-coloring” problem in the following way, which is close to the `DATALOGCIRC` program of Section 2.2 (in this case, the user must specify the name of the colors).

SPECIFICATION

```
Subset(NODE >< {red, green, blue}, coloring).
fail <-- edge(X,Y), coloring(X,C), coloring(Y,C).
fail <-- coloring(X,C1), coloring(X,C2), C1 /= C2.
colored(X) <-- coloring(X,_).
fail <-- node(X), co_colored(X).
```

The symbol “><” denotes the Cartesian product, “/=” is the *not equal* predicate, “_” is a mute variable, and `co_colored` is the complement of the predicate `colored`.

The output of the program for the database of Section 1 is in this case the following.

```
coloring: (1,blue), (2,green), (3,red), (4,blue), (5,green),
          (6,blue)
```

The specification is correct even if we drop the second rule. In this case, a node is allowed to have more than one color, and a solution can then be found by choosing arbitrarily any of its colors for each node.

3.2 Extensions of NP-SPEC: Tailoring Predicates

It is clear that, using the specification language described above, the generated program must make use of a pure “guess-and-check” mechanism, that explores all instances of Q predicates.

We note that, depending on the specific problem under consideration, it might be helpful to exclude some of the instances of the predicates in Q because

they are “structurally” infeasible. For example, it is possible for a given problem that the elements to be taken into account are all the permutations of a given set S , rather than the n -wise (n being equal to $|S|$) Cartesian product $S \times \dots \times S$. This happens, as an example, for the N-queens problem, where the search space is a permutation of the set $\{1, \dots, N\}$.

A search space structured as a family of permutations would in general be searched more efficiently than one deriving from a guess-and-check carried out on a full Cartesian product of the involved domains. One way to represent permutations is to use specification clauses that invalidate inappropriate instances. For example, we may use a dyadic predicate **Permute** to represent all possible permutations of S : The first attribute represents a generic element of S and the second one the position it holds in the permutation. According to the semantics, all subsets of $S \times \{1, \dots, |S|\}$ are possible extensions of **Permute**, but those that do not associate a position to an element, or associate more than one element to a position, do not correspond to a feasible element of the search space. Such elements could then be eliminated by using suitable clauses with **Permute** in the body and the atom **fail** in the head, such as the following one:

```
fail <-- Permute(X,I), Permute(Y,I), X /= Y.
```

Such an approach, however, has the drawback of requiring the analysis of *all* $(P; Q)$ -minimal models, which can outnumber the cardinality of the intended search space of the problem.

In order to overcome this drawback we introduce, in addition to **Subset**, more metapredicates, called *tailoring* predicates, that are used to specify that for a given predicate it is sufficient to consider only a certain class of extensions.

In the current implementation, the following metapredicates can be used:

- **Partition**(<DOMAIN>, <predicate_name>, n).
- **Permutation**(<DOMAIN>, <predicate_name>).
- **IntFunc**(<DOMAIN>, <predicate_name>, $\text{min}.. \text{max}$).

where <DOMAIN> is a domain as specified in Section 3.1 and <predicate_name> is a predicate in Q .

The metapredicate **Partition** has a further integer-valued argument n that states the number of subsets in which the domain must be partitioned. The metapredicate **IntFunc** has a composite argument $\text{min}.. \text{max}$ that determine the codomain $[\text{min}, \dots, \text{max}]$ of the integer function.

The predicate <predicate_name> should code the sets, or the collection of sets, built from <DOMAIN> according to the specific metapredicate used. Such coding is implemented using an extra internal domain, identified with an interval of integer values, called “ID domain” that has a different meaning for each metapredicate. The arity of <predicate_name> is augmented by one with respect to the arity of <DOMAIN>, in order to add an argument that ranges upon the ID domain.

As an example, if <DOMAIN> is $D = \{a, b, c, d\}$, the argument n for **Partition** is 3, and min and max are 1 and 30, respectively, the following are possible extension of the predicate used as argument of the metapredicate.

DOMAIN	<table><tr><th>IT</th></tr><tr><td>a</td></tr><tr><td>b</td></tr><tr><td>c</td></tr><tr><td>d</td></tr></table>	IT	a	b	c	d	PART	<table><tr><th>IT</th><th>ID</th></tr><tr><td>a</td><td>1</td></tr><tr><td>b</td><td>1</td></tr><tr><td>c</td><td>0</td></tr><tr><td>d</td><td>2</td></tr></table>	IT	ID	a	1	b	1	c	0	d	2	PERM	<table><tr><th>IT</th><th>ID</th></tr><tr><td>a</td><td>2</td></tr><tr><td>b</td><td>0</td></tr><tr><td>c</td><td>1</td></tr><tr><td>d</td><td>3</td></tr></table>	IT	ID	a	2	b	0	c	1	d	3	INTF	<table><tr><th>IT</th><th>ID</th></tr><tr><td>a</td><td>27</td></tr><tr><td>b</td><td>5</td></tr><tr><td>c</td><td>15</td></tr><tr><td>d</td><td>9</td></tr></table>	IT	ID	a	27	b	5	c	15	d	9
IT																																										
a																																										
b																																										
c																																										
d																																										
IT	ID																																									
a	1																																									
b	1																																									
c	0																																									
d	2																																									
IT	ID																																									
a	2																																									
b	0																																									
c	1																																									
d	3																																									
IT	ID																																									
a	27																																									
b	5																																									
c	15																																									
d	9																																									

The above instances represent, respectively, the partition of D into the three subsets $\{c\}, \{a, b\}, \{d\}$ of D , the permutation $[b, c, a, d]$ of D and the integer function f such that $f(a) = 27, f(b) = 5, f(c) = 15, f(d) = 9$.

For the sake of brevity, we do not describe other features of the language NP-SPEC which improve its usability.

3.3 The NP-SPEC Prototype

The NP-SPEC prototype is implemented mostly in SWI-PROLOG. The core of the system is composed of two modules. The first one, which is independent from the specification under consideration, implements the enumeration algorithm for the solution and takes care of the tailoring predicates and other secondary features of the language (e.g., the relational operators). The second module parses the specification and generates a PROLOG “translation” of it. The two modules are then compiled together automatically.

4 Specification Examples

In this section we illustrate the specification in NP-SPEC of two classical NP-complete problems, namely *Integer knapsack* [GJ79, problem MP10, page 247] and *Traveling salesman* [GJ79, problem ND22, page 211].

4.1 Integer Knapsack

It is given a *knapsack* of fixed capacity B , and a set of objects with their associated size and value. The *integer knapsack problem* consists in finding a selection of the given objects so that the capacity of the knapsack is not exceeded and a given minimum total value K is reached.

Differently from the simple knapsack problem, each object is available in unlimited quantity, therefore multiple copies of any object can be included in the solution.

The underlying NP-complete decision problem requires to decide whether such an assignment exists, and its mathematical specification is as follows, where $c(u)$ is the number of copies of the object u that are inserted in the knapsack.

INSTANCE: Finite set U , for each $u \in U$, a size $s(u) \in \mathbb{Z}^+$ and a value $v(u) \in \mathbb{Z}^+$, and positive integers B and K

QUESTION: Is there an assignment of a non-negative integer $c(u)$ to each $u \in U$ such that $\sum_{u \in U} c(u)s(u) \leq B$ and such that $\sum_{u \in U} c(u)v(u) \geq K$?

In NP-SPEC we write this specification in the following way

SPECIFICATION

```
IntFunc(U,take_up,0..B).
    // the max number of copies of any object is always
    // smaller than B
table(Obj,Value,Size) <-- take_up(Obj,Num), data(Obj,V,S),
                           Value == Num * V, Size == Num * S.
fail <-- SUM(table,3,S), S > B.
fail <-- SUM(table,2,V), V < K.
```

where the relation **take_up** represents the assignment c , and the three arguments of the ternary relation **data** encode the objects, their value, and their size, respectively.

The metapredicate **SUM** provides the sum of the elements of the projection on a given attribute of the relation passed as its first argument. The attribute to project on is provided by the second argument of **SUM** and the sum is delivered through the third argument.

A possible problem instance is described by the following **DATABASE** section of the specification.

DATABASE

```
U = {a,b,c,d,e};
DATA = {(a,8,15), (b,5,7), (c,7,11), (d,3,4), (e,6,10)};
    // encoding is (u, v(u), s(u))
B = 135;
K = 210;
```

Notice that the most natural way to specify this problem turned out to be the tailoring predicate **IntFunc**, rather than **Partition** used in Section 3.1 for the 3-coloring problem. In the next example the most suitable one is the predicate **Permutation**.

4.2 Traveling Salesman

The famous traveling salesman problem consists in finding a route that passes through all cities of a given map and returns in the initial one, which has a total length shorter than a given value B .

Here is the corresponding mathematical specification.

INSTANCE: Set $C = \{c_0, \dots, c_m\}$ of cities, distance $d(c_i, c_j) \in \mathbb{Z}^+$,
for each pair of cities $c_i, c_j \in C$, a positive integer B

QUESTION: Is there a tour of C having length B or less,
i.e., a permutation $\langle c_{\pi(0)}, \dots, c_{\pi(m)} \rangle$ of C such that

$$\left[\sum_{i=0}^{m-1} d(c_{\pi(i)}, c_{\pi(i+1)}) \right] + d(c_{\pi(m)}, c_{\pi(0)}) \leq B?$$

The distance among cities is represented by the function d which is not necessarily symmetric. In this case, it is natural to make use of the tailoring predicate `Permutation`.

DATABASE

```
C = {london,rome,paris,amsterdam};
D = {(london,rome,3125), (rome,london,3125),
      (london,paris,1230), (paris,london,1230),
      (london,amsterdam,800), (amsterdam,london,800),
      (rome,paris,2370), (paris,rome,2370),
      (rome,amsterdam,2000), (amsterdam,rome,2000),
      (paris,amsterdam,700), (amsterdam,paris,700)};
// encoding (ci,cj,d(ci,cj))
B = 6500;
m = 3; // there are m+1 cities to be visited
```

SPECIFICATION

```
Permutation(C,tour).
length(C,D) <-- tour(C,0), tour(C',m), d(C',C,D).
length(C,D) <-- tour(C,N), N' == N - 1, tour(C',N'),
                  length(C',D'), d(C',C,D''), D == D' + D''.
fail <-- tour(C,m), length(C,K), K > B.
```

Here, the atom `length(c,d)` holds if d is the overall distance from the city $c_{\pi(m)}$ to the city c in the tour as guessed in the predicate `tour`. For instance, if `tour` is $\{(london,0), (rome,1), (amsterdam,2), (paris,3)\}$ then `length` holds on the tuples $(london,1230), (rome,4355), (amsterdam,6355), (paris,7055)$.

Before closing the section, we note that usage of arithmetic can in principle lead to an increase in data complexity, thus exceeding NP. Anyway we note that NP remains the upper bound if the size of each arithmetical computation is polynomial in the size of the input database, as indeed happens in the examples presented in this section.

5 Considerations on Performance

In this work we mostly focus on the language design rather than on the efficiency of the programs generated. In fact, the current version of the system is meant only for developing executable specifications, and not for effective program synthesis. Nevertheless, in this section we provide an experimental analysis of its performances in order to assess how it compares to state-of-the-art solvers and algorithms.

We tested the performance of our prototype by solving some randomly generated instances of the 3SAT problem [GJ79, problem LO1, page 259], i.e., the satisfiability problem for propositional formulae in CNF when all clauses have

# variables	# clauses	ratio	% true	time (secs.)
10	48	4.8	49	1.19
11	53	4.82	51	2.35
12	58	4.83	56	4.32
13	62	4.77	49	8.87
14	68	4.86	56	17.80
15	70	4.67	49	36.97
16	76	4.75	42	77.72
17	79	4.65	56	151.65

Table 1. Average time for solving an instance of 3SAT close to the crossover point

exactly three literals. We performed our tests on a Sparc Ultra 2 running at 200 MHz.

It is known from the literature (cf. e.g., [SML96]) that, for many complete algorithms for the SAT problem, the hardest instances are close to the so-called “crossover point”, i.e., the point in which the probability of a randomly generated formula to be satisfiable is 50%. For such a reason we generated instances close to the crossover point. Table 1 shows the results of the experiment (average over 100 instances). Same data are reported on Figure 1, which highlights the exponential growth of the computing time.

For under-constrained instances of 3SAT, finding a solution is simpler, because typically a formula has many models. Our program was able to find a model of a formula with 50 variables and 60 clauses in about two hours.

State-of-the-art SAT solvers are obviously much faster. As an example the “Böhm solver”, an optimized version of the Davis-Putnam [DP60] procedure written in C and described in [BB93, Böb92], solves instances with 250 variables and 1075 instances in about 20 seconds on our machine.

Anyway, comparing our prototype to programs which implement similar algorithms gives a rosier picture. We tested the performance of an ad-hoc pure guess-and-check SAT solver we wrote in C++, and found that in 30 seconds it was able to solve instances with just 22 variables and 100 clauses (53% of true instances).

We hope that the prototype can be made more efficient by compiling the specification in a procedural language such as C/C++ and/or by generating programs that do not adopt a blind guess-and-check strategy, but rather make use of partial solutions and heuristics.

6 Related Work

The language NP-SPEC is similar in spirit to the system KIDS [Smi90], which produces the concrete implementation starting from the specification written in a logic language based on set theory. The main characteristic of KIDS is the use of a “domain theory” (written in terms of a set of axioms), which guides the

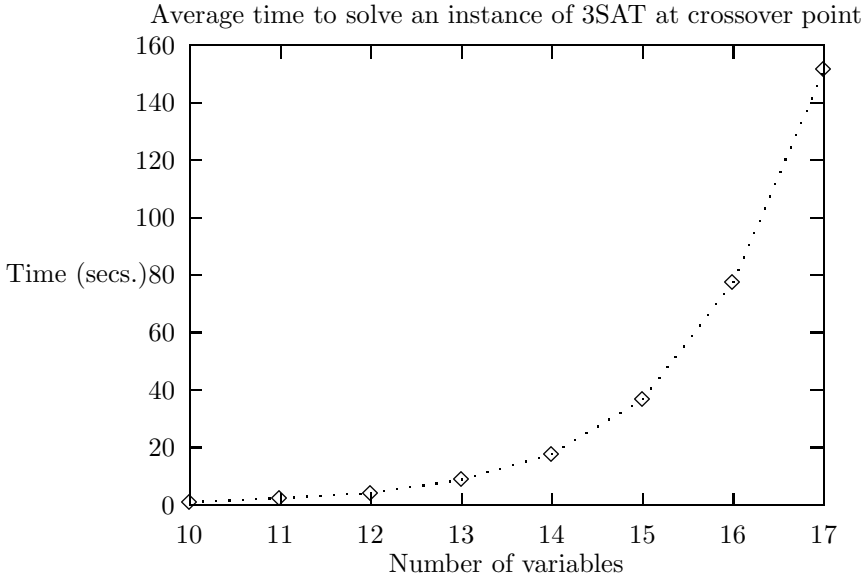


Fig. 1. Exponential growth of the computing time

system in the application of a predefined set of transformation rules that lead to the synthesis of a program compliant to the specified strategy.

Similar to our proposal, KIDS makes use of an abstract specification language “sensible” to the implementation issues. Moreover, in both languages, it is present the idea of selecting the search space which is the most suitable for the specific problem. In NP-SPEC, however, the concept of domain theory is absent, and this is due to the lack of a deductive mechanism. In fact, the translation of an NP-SPEC specification is done in a single step, and this is made possible by the relative limitedness of the expressiveness of NP-SPEC. Such limitation, which is characterized in an exact formal way (see Section 2.3), is one of the strengths of NP-SPEC, together with the capability of a fully automatic translation (in KIDS a certain degree of interactivity is required).

Another specification language proposed in the literature is SPILL [KM97]. In few words, SPILL is a typed subset of pure PROLOG. The main limitation with respect to PROLOG is the so-called *groundness restriction*: at run time, terms must not contain variables and they must be finite. However, differently from NP-SPEC, SPILL allows the use of function symbols in the language. As another difference, its semantics is a pure first order one, as it does not include any sort of model minimization operations.

A specification in SPILL is also “executable”, but not in the same sense of NP-SPEC. That is, in SPILL it is not possible to compute a solution of a problem from the specification, but rather only to *test* whether a provided solution is feasible, according to the specification. This limitation is a precise design deci-

sion, because SPILL, different from NP-SPEC, is not meant for computing, but to test the specification against some specific “interesting” cases. Finally, like KIDS, SPILL does not provide a characterization of its expressiveness and its complexity.

Another remarkable specification language is the one proposed by Minton [Min96] for the MULTI-TAC system. Such language is a sorted first order logic, which is specifically oriented to the specification of a problem by means of constraints. As for the other languages, no characterization is provided in the paper for the expressiveness of the language.

Finally, we note that there has been recently considerable interest in the implementation of rule-based deductive systems which semantics rely on non-monotonic formalisms (cf. e.g., [ELM⁺98]).

7 Conclusions and Future Work

We have presented NP-SPEC, an executable specification language for search problems, and we have shown how some classical problems can be expressed in NP-SPEC in a natural and concise way.

Even in the limited present form, with this small number of metapredicates, we have been able to specify in NP-SPEC a number of problems in NP in a simple way, the most significant of which are (in square parentheses we have reported the reference number in the Garey and Johnson’s list [GJ79]): *Inequivalence of simple functions* [PO15]; *Register sufficiency* [PO1]; *Dynamic storage allocation* [SR2]; *Pruned trie space minimization* [SR3]; *Integral flow with multipliers* [ND33]; *Consecutive ones matrix augmentation* [SR16]; *Boyce-Codd normal form violation* [SR29]; *Quadratic Diophantine equations* [AN8]; *Generalized instant insanity* [GP15]; *Modal logic S5 satisfiability* [LO13]; *Code generation for parallel assignments* [PO6]; *Balanced complete bipartite subgraph* [GP24]; *Intersection graph basis* [GT59]; *Minimum broadcast time* [ND49].

Differently from most specification languages in the literature, NP-SPEC has a precise characterization of its expressive power, namely the class NP. On the one hand, such expressiveness guarantees to the user the decidability of the execution and, to a limited extent, its efficiency. On the other hand, it allows the designer of the system to implement optimized solution techniques.

Regarding the optimization of the execution, in the future we plan to work primarily in two directions:

- Improve the language extensions of NP-SPEC by including a limited number of additional metapredicates. The need for them might indeed arise from further experimentations with the language. This will be done with the twofold objective of allowing more natural specifications and improve the efficiency of their execution.
- Provide more efficient execution code by extracting further information from the specifications, mainly by means of a static analysis of rules. This should lead to smarter, backtracking-based execution strategies (e.g., backjumping, forward-checking) than pure enumeration.

As of the latter point, we are also planning to re-implement the system by using a constraint language such as, for example ILOG Solver [ILO98] or *ECLⁱPS^e* [Agg98].

Acknowledgments

The first author has been in part supported by a scholarship from the Italian National Research Council (CNR) under the “Short-term mobility” program. The work has also been supported by ASI (Italian Space Agency) and MURST (Italian Ministry for University and Scientific and Technological Research) under the 40% “Interdata” project.

References

- [Agg98] A. Aggoun *et al.* *ECLⁱPS^e User Manual (Version 4.0)*. IC-Parc, Germany, July 1998.
- [Böh92] M. Böhm. Sat solver, 1992. Computer program available at www.informatik.uni-koeln.de/lj_s_juenger/staff/boehm/src.html.
- [BB93] M. Buro and H. Kleine Büning. Report on a SAT competition. *EATCS Bulletin*, 49:143–151, 1993.
- [CP98] M. Cadoli and L. Palopoli. Circumscribing DATALOG: expressive power and complexity. *Theoretical Computer Science*, 193:215–244, 1998.
- [DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.
- [ELM⁺98] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR system dl_v: Progress report, comparisons and benchmarks. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR-98)*, pages 406–417, 1998.
- [GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, Ca, 1979.
- [ILO98] ILOG optimization suite — white paper. Available at www.ilog.com, 1998.
- [KM97] F. Kluźniak and M. Milkowska. Spill – a logic language for writing testable requirements specifications. *Science of Computer Programming*, 28:193–223, 1997.
- [Lif85] V. Lifschitz. Computing circumscription. In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI-85)*, pages 121–127, 1985.
- [McC80] J. McCarthy. Circumscription - A form of non-monotonic reasoning. *Artificial Intelligence*, 13:27–39, 1980.
- [Min96] S. Minton. Automatic configuring constraint satisfaction programs: A case study. *Constraints*, 1(1/2):7–43, 1996.
- [Smi90] D. R. Smith. KIDS: A semi-automatic program development system. *IEEE Transactions on Software Engineering*, 16(9), 1990.
- [SML96] B. Selman, D. Mitchell, and H. Levesque. Generating Hard Satisfiability Problems. *Artificial Intelligence*, 81:17–29, 1996.
- [Ull88] J. D. Ullman. *Principles of Database and Knowledge Base Systems*, volume 1. Computer Science Press, 1988.
- [vEK76] M. H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23(4):733–742, 1976.

Prototyping a Requirements Specification through an Automatically Generated Concurrent Logic Program^{*}

Patricio Letelier, Pedro Sánchez, and Isidro Ramos

Department of Information Systems and Computation
Valencia University of Technology, 46020 Valencia (Spain)
{letelier, ppalma, iramos}@dsic.upv.es

Abstract. OASIS is a formal approach for the specification of object oriented conceptual models. In OASIS conceptual schemas of information systems are represented as societies of interacting concurrent objects. Animating such models in order to validate the specification of information systems is a topic of interest in requirements engineering. Thus a basic execution model for OASIS specifications has been developed. Concurrent Logic Programming is a suitable paradigm for distributed computation allowing a natural representation of concurrence. Using Concurrent Logic Programming, OASIS specifications are animated according to OASIS execution model. In this work, we show how OASIS concepts are directly mapped into concurrent logic programming concepts. To illustrate our ideas, an example of a bank account codified in KL1 is given and parts of the program that animates its corresponding OASIS specification are shown. This work has been developed in the context of a CASE tool supporting the OASIS approach. Our aim is to build a module for animation and validation of specifications. A preliminary version of this module is presented.

1 Introduction

Conceptual models, representing the functional requirements of information systems, are a key factor when linking the problem and solution domains. Building a conceptual model is a discovery process, not only for the analyst but also for the stakeholders. The most suitable strategy in this situation is to build the conceptual model in an iterative and incremental way, through analyst and stakeholder interaction. Conceptual modeling involve four activities: elicitation of requirements, modeling or specification, verification of quality and consistency, and eventually, validation.

Formal methods for conceptual modeling provide improvements in soundness and precision for specifications, simplifying their verification. However, when

^{*} This research is supported by the “Comisión Interministerial de Ciencia y Tecnología” (CICYT) through the MENHIR proyect (grant no. TIC97-0593-C05-01).

considering elicitation and requirements validation, prototyping techniques are more used. Hence, it is interesting to obtain a combination of both approaches.

This work uses OASIS [9,13] (Open and Active Specification of Information Systems) as a formal approach for object-oriented conceptual specification of information systems. This is a step forward in a growing research field where validation of formal specifications through animation is being explored [18]. In this sense, some other proposals close in nature to OASIS are [7] and [6]. The differences, though, between these works and ours are basically determined by features of the underlying formalisms and the offered expressiveness. According to the presented results, the state of the art is similar and is characterized by preliminary versions of animation environments.

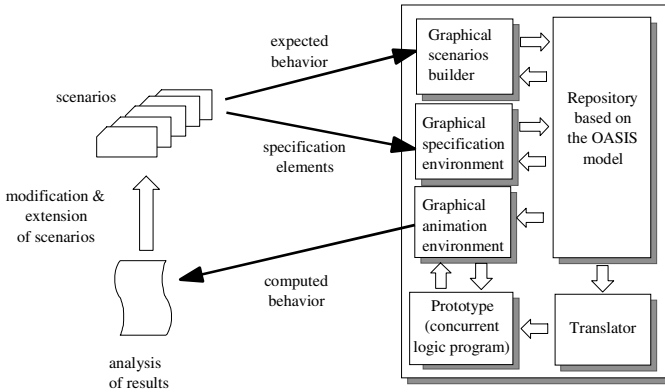


Fig. 1. A framework for incremental specification of requirements

Fig. 1 shows our framework for elicitation, modeling, verification and validation of requirements. Elicitation is achieved using scenarios [15]. The expected behavior and elements of a given specification are extracted from scenarios. The graphical scenario builder helps defining scenarios in a suitable way. Functional requirements are modeled using a graphical specification module based on OASIS. Conceptual models can be verified according to OASIS formal properties. At each stage of the requirements specification process it would be possible to validate the behavior of the associated prototype against the expected behavior. This comparison could lead to updates or extensions of existing scenarios. This cycle continues until the requirements are compliant with the proposed set of scenarios.

Experiments have been carried out using Object Petri Nets [16] and Concurrent Logic Programming [8] as semantic domains for OASIS specifications. These efforts have resulted in a basic execution model [10]. This model is used to animate OASIS specifications implemented over concurrent programming environments.

In this context, we will present the mappings between OASIS concepts and clauses in concurrent logic programming. Those mappings have been implemented in a translator program. The translator takes an OASIS specification stored in the repository and generates automatically a concurrent logic program that constitutes a prototype for the corresponding conceptual model. Furthermore, through a preliminary version of the graphical animation environment the analyst can interact with the prototype in a suitable way. We have worked with the concurrent logic languages Parlog [3] and KL1 [2] which have similar features. The implementation showed in this work is KL1 code.

2 OASIS

An OASIS specification is a presentation of a theory in the used formal system and is expressed as a structured set of class definitions. Classes can be simple or complex. A complex class is defined in terms of other classes (simple or complex). Complex classes are defined by establishing relationships among classes. These relationships can be aggregation or inheritance. A class has a name, one or more identification mechanisms for its instances (objects) and a type or template that is shared by every instance belonging to the class. We present next the basic concepts of OASIS.

Definition 1. *Template or type.* A class template is represented by a tuple $\langle \text{Attributes}, \text{Events}, \text{Formulae}, \text{Processes} \rangle$.

Attributes is the alphabet of attributes. For all $att \in \text{Attributes}$ exists the function:

$$att : \text{sort of names} \rightarrow \text{sort of values}$$

Events is the alphabet of events. For all $e \in \text{Events}$ is possible to get $\underline{e} = \theta e$ being θ a basic substitution of the parameters of the event. *Formulae* is a set of formulae which are organized in sections and their underlying formalism depends on the section where they are used. *Processes* is the set of process specifications, classified in protocols and operations.

Definition 2. *Service.* A service is either an event or an operation. The former is an instantaneous and atomic service. An operation is a non-atomic service and, in general, has duration.

Definition 3. *Action.* An action is a tuple $\langle \text{Client}, \text{Server}, \text{Service} \rangle$. It represents the action in the client object, associated to requiring the service, as well as the action in the server object, associated to providing the service.

For each class we assume the implicit existence of A , a set of actions obtained from the services that objects in the class can request (clients) or provide (servers). For all $a \in A$ is possible to obtain $\underline{a} = \theta a$, being θ a basic substitution of client, server and service.

Definition 4. *Object state.* An object state is a set of evaluated attributes. It is expressed by well-formed formulae in First Order Logic.

Definition 5. *Step.* A step is a set of actions occurring simultaneously in an object's life.

Definition 6. *Object life or trace.* An object's life or trace is a finite prefix of object steps.

2.1 OASIS Expressed in Dynamic Logic

In [11] Deontic Logic [1] is described as a variant of Dynamic Logic [5]. The definition of deontic operators in Dynamic Logic is:

$\psi \rightarrow [a]false$	“the occurrence of a is forbidden in states where ψ is satisfied”.
$\psi \rightarrow [\neg a]false$	“the occurrence of a is obligatory in states where ψ is satisfied”.
$\psi \rightarrow [a]\phi$	“in states where ψ is satisfied, immediately after of the a occurrence, ϕ must be satisfied”

where ψ is a well-formed formulae that characterizes an object's state when the action a occurs and $\neg a$ represents the non-occurrence of the action a (i.e., only other actions different from a could occur). Furthermore, there is no state satisfying the atom *false*. This represents a state of system violation. Thus, one action is forbidden if its occurrence leads the system towards a violation state, and one action is obligatory if its non-occurrence leads the system towards a violation state. The OASIS *Formulae* and *Processes* are mapped to the formulae previously presented.

These formulae constitute a sublanguage of the language proposed and formalized in [20]. In [9] OASIS is presented as a specification language with a well defined syntax. Here is an example of part of a simple bank system using the OASIS syntax. This example will be used in the rest of the paper.

```
conceptual schema simple_banking_system
class account
identification
    number:(number);
constant attributes
    number:nat; name:string;
variable attributes
    balance:nat(0); times:nat(0); pin:nat(0); rank:nat(0);
derived attributes
    good_balance:bool;
derivations
    good_balance:={balance>=100};
```

```

events
    open new; close destroy;
    deposit(Amount:nat);
    withdraw(Pin:nat,Amount:nat);
    pay_commission;
    change_pin(Pin:nat,NewPin:nat);
    change_rank(Rank:nat);
valuations
    [deposit(Amount)] balance:=balance+Amount, times:=times+1;
    [withdraw(Pin,Amount)] balance:=balance-Amount, times:=times+1;
    [self:pay_commission] balance:=balance-1;
    [self:pay_commission] times:=0;
    [change_pin(Pin,NewPin)] pin:=NewPin;
    [change_rank(Rank)] rank:=Rank;
preconditions
    withdraw(Pin,Amount) if (pin=Pin and balance>=Amount) or
                          (pin=Pin and balance<Amount and rank=2);
    change_pin(Pin,NewPin) if (pin=Pin);
    close if (balance=0);
triggers
    self::pay_commission when
        (times>=5 and good_balance=false and rank=0);
end class
class customer
identification
    name:(name);
constant attributes
    name:string;
events
    add new; remove destroy;
end class

interface customer(someone) with account(someone)
    services(deposit,withdraw,change_pin);
end interface

interface account(someone) with self
    services(pay_commission);
end interface
end conceptual schema

```

In this example there are two classes: **customer** and **account**. The objects in both classes are active. An **account** object is forced to self-trigger an action with the event **pay_commission** whenever the trigger condition is satisfied. Although **customer** objects do not have explicit triggers, they have an interface with **account** objects enabling to require actions associated with the visible events. Thus **customer** objects are active objects as well. Furthermore, by default, there is always an object called **root**. This object will be responsible for activating the events that do not have an explicit client in the specification. In the example, the **root** object can require actions with the event **add** and **remove** to the **customer**.

2.2 An Execution Model for OASIS

Our execution model is an abstract animator for formulae of obligation, permission and change of state associated to an object. Next we briefly describe the concepts included in the execution model proposed in [10].

The sequence of steps in an object's life is sorted by time. We assume there is a “clock object” sending special actions — called *ticks* — to every object in the system. The received *ticks* by an object are correlative with natural numbers, t_1, t_2 , etc. Hence, being i and j natural numbers then $i < j \iff t_i < t_j$.

Definition 7. Mailbox. *A mailbox is the set of actions that can be included in a step executed by one object at one tick. The mailbox at instant t_i is denoted by $Mbox_i$.*

Definition 8. Object's state at tick. *An object's state at tick is denoted by $State_i$ and represents the object's state in the interval $[t_i, t_{i+1})$. That is, between t_i (included) and t_{i+1} the state is considered constant.*

The processing of the actions inside the mailbox implies their classification. Next we present all possible actions that might be present in a mailbox. They are characterized as subsets of $Mbox_i$ (that is at instant t_i) .

- **Obligated actions:** these are actions associated to obligated service requests (which have as a client the object itself) and **have to** occur. The set of obligated actions is denoted by $OExec_i$. These actions are determined by obligation formulae, that is, their form is: $\phi[-a]false$.
- **Non-obligated actions:** these are actions corresponding to services requested by other objects (or itself) which **could be** provided or not depending on prohibitions established over those actions and verified in $State_i$. The set of non-obligated actions is denoted by \overline{OExec}_i .
- **Rejected actions:** these are non-obligated actions whose occurrence is prohibited in $State_i$. The set of rejected actions is denoted by $Reject_i$. These actions are determined by prohibition formulae, that is, their form is: $\phi[a]false$.
- **Candidate actions:** these are non-obligated actions whose occurrence is permitted in $State_i$. The set of candidate actions is denoted by $Cand_i$.
- **Executed actions:** these are actions forming the step executed at t_i . The set of executed actions is denoted by $Exec_i$. A step is composed by $OExec_i$ joined with a subset of $Cand_i$.
- **Actions in conflict:** These are a subset of $Cand_i$ formed by actions in conflict¹ with some obligated or candidate action just selected. The set of actions in conflict is denoted by $Conf_i$.

A simple criterion is used in order to choose actions from $Cand_i$: when two actions are in conflict, the action which first arrived to the mailbox will be chosen.

¹ Two actions are in conflict if they could change the value of non-disjoint set of attributes.

The actions in conflict $Conf_i$ are “copied” to the next mailbox ($Mbox_{i+1}$). The object behavior is characterized by an algorithm (detailed in [10]) that manipulates each mailbox (at each *tick*) obtaining the subsets previously defined and producing the change of the object state.

3 Concurrent Logic Programming and OASIS

Concurrent logic languages arise as an attempt to improve the efficiency of logic languages by exploiting the stream AND parallelism. Besides, they are high level programming languages and very convenient for parallel and distributed systems. A concurrent logic program is a set of Horn Clauses with Guards.

$$H \leftarrow G_1, \dots, G_n : B_1, \dots, B_m \quad n, m \geq 0$$

A goal has the following form:

$$\leftarrow M_1, \dots, M_k \quad k > 0$$

All M_i are evaluated in parallel (AND parallelism), using the program clauses for their reduction. For each M_i the clauses that can reduce it are examined in parallel (OR parallelism), selecting only one and avoiding backtracking. The criterion of selection is that the M_i unifies with the head of the clause, and the conjunction of guards G_1, \dots, G_n will be satisfied (evaluated also using AND parallelism, if they exist). If more than one clause could be chosen, then a sequential search can be established in textual order from top to bottom.

The integration of Concurrent Logic Programming and the OO modeling has generated a great deal of research. We are interested in modeling objects as perpetual processes according to the OASIS execution model. Modeling objects as perpetual processes is an approach initiate by Shapiro and Takeuchi [17], in which an object is implemented as a tail-recursive process that passes the update state of the object as arguments in the recursive call. The identity of the object is the name of an input stream argument of the process. Works in this direction are principally based on making OO extensions to concurrent logic languages. Within this approach some proposals are: Polka [4], L2||O2 [14] and A’UM [19]. Although some implementation aspects are common, our motivation is to generate automatically a concurrent logic program corresponding to an OASIS conceptual model.

3.1 Objects and Classes in Concurrent Logic Programming

Now we will sketch the essential features that allow considering an OASIS specification as a KL1 program. Details of intermediate clauses and clause bodies will be omitted to facilitate their reading. In Concurrent Logic Programming a

society of objects can be seen (at run time) as a goal to solve, where each object is a subgoal. Each object is evaluated using AND parallelism (inter-object concurrence).

$$\leftarrow object_1(In_1, Out_1, State_1), \dots, object_k(In_k, Out_k, State_k) \quad k > 0$$

In Concurrent Logic Programming, the partial instantiation of logical variables enables to use them as communication channels. In_i is the input channel for receiving actions, it is a merger of the messages received from itself and from other objects. Also, In_i is used as the object *oid*, so anybody knowing the value of *oid* could instantiate partially this variable, that is, send an action to that object. Out_i is the output channel for sending actions. $State_i$ is the list of terms $att(Attributte, value)$ for each attribute of the object.

In general, creation (and destruction) of objects at run time is required. Subgoals that have the capacity of generating (in their reduction) a new instance of an object should exist. These subgoals correspond in a natural form to the notion of class. Thus, classes are implemented as other objects of the society. A class has an attribute called *Population*, which is a list of pairs (In_i, Key) for each class instance. In_i is the object *Oid*, *Key* is a list of constant attributes that allows referring to the object. The predicate name of a class goal will be the name of the corresponding class. For an object goal the predicate name will be the name of its class with “o_” at the beginning.

Example 1. The class **account** in our bank system (at run time) is represented as a goal in Concurrent Logic Programming. Classes are goals at the beginning of the execution. Thus the class **account** appears as the following subgoal:

`:- ..., account(In, Out, [att(population, [])]), ...`

3.2 Object Behavior

Each object attempts to reduce itself using the clauses that represent its specification. Each clause recognizes one action and is able to reduce the object to a set of subgoals. Below an object goal and clauses with which this object goal could be reduced are shown. Considering $p, q, s, t \geq 0$ and $r > 0$:

$: - \dots, o_class_i(In^*, Out^*, State^*), \dots$

$o_class_i([action_{i1}|In], Out, State) : -$
 $G_{i1}, \dots, G_{ip} |$
 $B_{i1}, \dots, B_{it},$
 $o_class_i(In, NOut, NState).$


```

...
o_classi([actionir|In], Out, State) :-
    Gi1, ..., Giq |
    Bi1, ..., Bis,
    o_classi(In, NOut, NState).
    
```

In^* , Out^* and $State^*$ represent the logical variables In , Out and $State$ at a given moment in the reduction of the goal o_class_i .

Object Creation and Destruction Classes are implemented as subgoals inside the initial goal. We obtain the instances by means of reduction of class goals whenever the action occurs with the event of creation. The creation of an object implies that the following subgoals are generated during the class goal reduction:

```

...
NOut = {Out, ObjectOut},
o_class(Oid, ObjectOut, Attributes),
...
    
```

Out is the output channel for the class goal. This channel is separated into two new logical variables. $NOut$ will be used as the new output channel for the class goal. $ObjectOut$ will be the output channel for the object created. Oid is another new logical variable that will be used as object Oid and input channel.

Example 2. The clause to which the class `account` reduces when the action of creation occurs (action with the `open`² event). In the body of that clause there is the creation of a new object (goal) `account`.

```

account([action(Client,s(account),e(open,Attributes))|RestActions],
        Out,State) :-
    Out={NOut,ObjectOut},
    o_account(Oid,ObjectOut,Attributes),
    get_attributes(Attributes,[att(number,Number)]),
    NPopulation=[object(Oid,Number)|Population],
    update_state(State,[chg(population,NPopulation)],NState),
    ...
    account(RestActions,NOut,NState).
    
```

`get_attributes` is a predicate that extracts some attribute values from a list. `update_state` modifies a list of attributes producing a new one. When the event `open(101, john, 0, 0, 1234, 0, false)` is received the goal `account` becomes:

```

account(action(Client,c(account),
    e(open,[101,john,0,0,1234,0,false]))|Rest],Out,State)
    
```

This goal is reduced in the following two subgoals, that is, a new object `account` has been created.

² An event of creation has implicitly the constants and variable attributes of the object as arguments.

```

...
o_account(Objid,ObjectOut,[att(number,101),att(name,john),
    att(balance,0),att(times,0),att(pin,1234),
    att(rank,0),att(good_balance,false)]),
account(Rest,NOut,)[att(population,[object(Objid,101)])].

```

The destruction of an object is obtained by the reduction carried out by selecting a clause whose body does not contain the same object as subgoal. Hence, the execution of the object ends.

Change of State We say that an object is implemented as a perpetual process because among the subgoals in which an object is reduced the same object appears. This produces the effect of continuity in the object life. Whenever the object goal is thrown as subgoal in the reduction, some of its attributes may be modified. Thus a change of state due to the occurrence of the associate action is represented. The effect “to execute action” is obtained in the reduction when the new input channel is used as the original one without considering the last executed action. The formulae that define the change of state when the event is executed are subgoals that assign new values to the attributes of the object.

Example 3. The execution of the action associated with the event `deposit(10)` sent to the recently created object is represented as the reduction of the goal:

```

o_account([action(Client,s(account,id(number,[att(number,101)])),
    e(deposit,[10]))|RestIn],Out,[att(number,101),att(name,john),
    att(balance,0),att(times,0),att(pin,1234),att(rank,0),
    att(good_balance,false)])

```

This goal is reduced using the following clause:

```

o_account([action(Client,Server,e(deposit,[N]))|Rest],Out,State) :-
    get_attributes(State,[att(balance,Balance),att(times,Times)]),
    NBalance:= Balance+N,
    NTimes:= Times+1,
    LExp1:= NBalance,
    RExp1:= 100,
    test_condition([[c(ge,LExp1,RExp1)]],NGood_balance),
    update_state(State,[chg(balance,NBalance),chg(times,NTimes),
    chg(good_balance,NGood_balance)],NState)
o_account(Rest,Out,NState).

```

`test_condition` is a predicate that evaluates a list of conjunctions representing a well-formed formulae in the specification. If all the conjunctions are satisfied the second argument of `test_condition` is instantiated to *true*, otherwise it is *false*. Thus, in this case, the goal is reduced to the following goal, in this way, the object has changed its state.

```

o_account(Rest,Out,[att(number,101),att(name,john),att(balance,10),
    att(times,1),att(pin,1234),att(rank,0),att(good_balance,false)])

```

Prohibitions Action preconditions are implemented through intermediate predicates in the body of the object clauses.

Example 4. Here is the clause that verifies preconditions when the event `withdraw(1234,10)` is attended.

```
check_one_o_account_condition(action(Client,This,e(withdraw,[P,N])),
    State,Result,Msg) :-
    utility:get_attributes(State,[att(balance,Balance),
        att(rank,Rank),att(pin,Pin)]),
    LExp1:= Balance, RExp1:= N,
    LExp2:= Balance, RExp2:= N,
    LExp3:= Rank, RExp3:= 2,
    LExp4:= P, RExp4:= Pin,
    test_condition([[c(eq,LExp4,RExp4),c(ge,LExp1,RExp1)],
        [c(eq,LExp4,RExp4),c(lt,LExp2,RExp2),c(eq,LExp3,RExp3)]] ,
        Result),
    Msg=not([[c(eq,LExp4,RExp4),c(ge,LExp1,RExp1)],
        [c(eq,LExp4,RExp4),c(lt,LExp2,RExp2),c(eq,LExp3,RExp3)]]).
```

This time, the predicate `test_condition` is used to instantiate to *true* or *false* the variable `Result`. When the precondition is not satisfied another action is sent to the client including the message `Msg`.

3.3 Inter-object Communication

The communication mechanism offered by Concurrent Logic Programming is to share variable among subgoals, interpreted as a communication channel among objects. In order to allow many objects to communicate with a determined object, the architecture showed in Fig. 2 has been implemented.

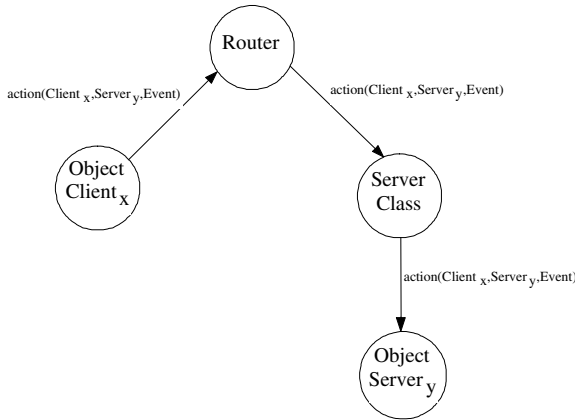


Fig. 2. Communication between objects

In general, if a client object $client_x$ has to send an action to a server object (or itself) $server_y$, $client_x$ instantiates partially its variable *Out*. The *Out* variable is merged with the rest of the output variables of objects and is used as input variable for an additional goal called *router*. The *router* goal has as many output variables as classes in the system. Whenever the goal *router* receives an action in its input, it will put the action in the corresponding output channel according to the server of the action.

The first goal in a KL1 program is the atom *main*. Thus, in the body of this predicate the global schema of communication is established.

Example 5. The main predicate defined for our bank system.

```
main:-
  take_active_actions(ExternActions),
  UserActions=[action(c(tester),s(user),e(add,[att(userid,root)]))
    |InUser],
  user(InUser,OutUser,[att(population,[])]),
  customer(InCustomer,OutCustomer,[att(population,[])]),
  account(InAccount,OutAccount,[att(population,[])]),
  generic:new(merge,{ExternActions,OutUser,OutCustomer,OutAccount},
    Actions),
  router(Actions,InUser,InCustomer,InAccount).
```

take_active_actions is a predicate that takes the external actions during the session of animation. There is always a class called *user* and an object of this class whose attribute *userid* has the value *root*.

Obligations In OASIS the object communication is activated by obligations. In this work we will focus on the fundamental communication mechanism, represented by triggers in the OASIS syntax. Triggers represent asynchronous communication. Hence, triggers are implemented executing $Out = [Action|NOut]$ without waiting for an answer. Triggers are generated due to a change of state. Thus, in the body of each clause that represents a change of state related with a condition in the trigger, a goal that evaluates the condition and could shoot the trigger will be added. This sentence will appear in the body of a selected clause in the reduction of a client object. *Out* is the identifier of the shared variable that constitutes the output channel of actions for the client object. This is a term containing the client reference. *NOut* is the new variable for the output channel of the client object.

Actions must first arrive to the router that sends the action to the corresponding input channel of class. If the action server is a class instance, the associated class searches in its *Population* attribute the input channel of the server object.

Example 6. A goal representing an object *account* reaching a new state:

```
o_account(In,Out,[att(number,101),att(name,john),att(balance,80),
  att(times,5),att(pin,1234),att(rank,0),att(good_balance,false)])
```

The possibility of trigger activation in the new state (that will be reached) was detected during the previous change of state. The clause doing this work is:

```
verify_o_account.triggers([t1|RestTriggers],State,Triggers) :-
    utility:get_attributes(State,[att(times,Times),
                                att(good_balance,Good_balance),att(rank,Rank)]),
    LExp1:= Times,    RExp1:= 5,
    LExp2=Good_balance, RExp2=false,
    LExp3:= Rank,    RExp3:= 0,
    utility:test_condition([[c(ge,LExp1,RExp1),
                            c(eq,LExp2,RExp2),c(eq,LExp3,RExp3)]] ,Answer),
    utility:get_attributes(State,[att(number,Number)]),
    utility:put_trigger(Answer,action(this,self,e(pay_commission,[])),
                        Triggers,NTriggers),
    verify_o_account.triggers(RestTriggers,State,NTriggers).
```

`put_trigger` is a predicate that puts the action in the trigger list if `Answer` is instantiated to *true*. In this case because `test_condition` returns *true* in `Answer`, we are sure that the object *Mbox* contains the action:

```
action(this,self,e(pay_commission,[]))
```

This is an action that must be triggered because the client term is `this`. Thus it is put in the output channel using:

```
Out=[action(This,self,e(pay_commission,[]))|NOut]
```

Now `This` is the client specification referring to its identification.

4 A Graphical Animation Environment

In the OASIS context, the system behavior is determined by the behavior of its objects. An object behavior can be observed by analyzing the actions occurred and the states reached by the object. In this sense, the animation of an OASIS specification allows examining actions and states of the objects. Following the guides mentioned before, a translator from OASIS to KL1 has been implemented. This translator takes as input a system specification from an OASIS repository and produces a KL1 program that is compiled in order to obtain the prototype. The translator and the prototype are programs running in a Unix workstation. The interface has been implemented in Tcl/Tk [12] using the Tcl plug-in for Netscape and establishing a socket connection to the Unix workstation.

Fig. 3 shows the interface for a preliminary version for an environment of animation. The idea is to make easier the use of the prototype. The object society is drawn in the upper left corner, on the right the traces of actions of an object (or object group) are listed. The list of traces can be filtered according to the kind of actions defined (*OExec_i*, *OExec_i*, *Conf_i*, *Exec_i* and *Rejected_i*). In the state area the state of the object is presented (only when one object is selected). Buttons *play*, *pause*, *stop*, *forward* and *review* are provided in order to control the session of animation. When the animation is paused it is possible to explore the traces of actions and states at previous instants. Eventually, the two entry widgets allow building an external action sent by the analyst in representation of one object in the system.



Fig. 3. An animation session

5 Conclusions

We have shown how the main features of OASIS can be naturally and directly represented in a concurrent logic program. Using the execution model of OASIS as a guide we can obtain a useful animation of the OASIS specification. Our animation is only applied to purposes of requirements validation and do not claim to be the final software product. The fidelity of the obtained concurrent logic program in relation to the OASIS system specification is a matter that is still being studied. In this case the verification and demonstration tasks would be supported by three important factors: firstly at the conceptual level the model is described in a formal language, secondly the abstract execution model is inspired by the semantics of that formal language and eventually there are some proposals dealing with formalization of concurrent programming languages. These factors do not determine the required justification but give a way of formalization on which we are working.

We have built a translator program to obtain a concurrent logic program automatically from an OASIS specification using the established correspondences. This work is being integrated into a CASE tool for system modeling supporting the OASIS model.

References

1. L. Åqvist. Deontic logic. In D.M. Gabbay and F.Guenther, editors, *Handbook of Philosophical Logic II*, pages 605-714. Reidel, 1984.
2. T. Chikayama. *KLIC User's Manual*. Institute for New Generation Computer Technology, Tokyo JAPAN, 1995.
3. T. Conlon. *Programming in PARLOG*. Addison-Wesley, 1989.
4. A. Davison. *Polka: A Parlog object-oriented language*, Ph.D. thesis, Department of Computer Science, Imperial College London, 1989.
5. D. Harel. *Dynamic Logic*. In *Handbook of Philosophical Logic II*, editors D.M.Gabbay, F.Guenther; pages 497-694. Reidel 1984.
6. P. Heymans. *The Albert II Specification Animator*. Technical Report CREWS 97-13, Cooperative Requirements Engineering with Scenarios, <http://sunsite.informatik.rwth-aachen.de/CREWS/reports97.htm>.
7. R. Herzig and M. Gogolla. An animator for the object specification language TROLL light. In *Proc. Colloq. on Object-Orientation in Databases and Software Engineering*, Montreal 1994.
8. P. Letelier, P. Sánchez and I. Ramos. Animation of system specifications using concurrent logic programming. *Symposium on Logical Approaches to Agent Modeling and Design, ESSLLI'97*, Aix-en-Provence, France, 1997.
9. P. Letelier, I. Ramos, P. Sánchez and O. Pastor. OASIS 3.0: Un enfoque formal para el modelado conceptual orientado a objeto. SPUPV-98.4011, Servicio de Publicaciones Universidad Politécnica de Valencia, 1998.
10. P. Letelier, I. Ramos and P. Sánchez. Un modelo de ejecución para especificaciones OASIS 3.0 Informe Técnico DSIC-II/36/98, Universidad Politécnica de Valencia, 1998.
11. J.-J.Ch. Meyer. A different approach to deontic logic: Deontic logic viewed as a variant of dynamic logic. In *Notre Dame Journal of Formal Logic*, vol.29, pages 109-136, 1988.
12. J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.
13. O. Pastor and I. Ramos. OASIS version 2 (2.2) : A Class-Definition language to model information systems using an object-oriented approach, SPUPV-95.788, Servicio de Publicaciones Universidad Politécnica de Valencia, 1995.
14. E. Pimentel. *L2||O2: Un lenguaje lógico concurrente orientado a objetos*, Tesis Doctoral, Facultad de Informática, Universidad de Málaga, 1993.
15. C. Rolland, C. Ben Achour, C. Cauvet, J. Ralyté, A. Sutcliffe, N.A.M. Maiden, M. Jarke, P. Haumer, K. Pohl, E. Dubois and P. Heymans. A Proposal for a Scenario Classification Framework, Technical Report CREWS 96-01, <http://sunsite.informatik.rwth-aachen.de/CREWS/reports96.htm>.
16. P. Sánchez, P. Letelier and I. Ramos. Constructs for Prototyping Information Systems using Object Petri Nets, *Proc. of IEEE International Conference on System Man and Cybernetics*, pages 4260-4265, Orlando, USA, 1997.
17. E. Shapiro and A. Takeuchi. Object oriented programming in concurrent prolog, in *New Generation Computing*, vol.1, pages 25-48, 1983.
18. J. Siddiqi, I.C. Morrey, C.R. Roast and M.B. Ozcan. Towards quality requirements via animated formal specifications. *Annals of Software Engineering*, n.3, 1997.
19. K. Yoshida and T. Chikayama. A'UM: A string based concurrent object-oriented language, In *Proc. of the Int.Conf. on FGCS, ICOT*, pages 638-649, 1988.
20. R.J. Wieringa and J.-J.Ch. Meyer. Actors, Actions and Initiative in Normative System Specification. *Annals of Mathematics and Artificial Intelligence*, 7:289-346, 1993.

Multi-agent Systems Development as a Software Engineering Enterprise

Marco Bozzano¹, Giorgio Delzanno², Maurizio Martelli¹, Viviana Mascardi¹,
and Floriano Zini¹

¹ D.I.S.I. - Università di Genova

via Dodecaneso 35, 16146 Genova, Italy

{bozzano,martelli,mascardi,zini}@disi.unige.it

² Max-Planck Institut für Informatik

Im Stadtwald, Gebaude 46.1, D-66123 Saarbrücken

delzanno@mpi-sb.mpg.de

Abstract. Multi-Agent Systems provide an ideal level of abstraction for modelling complex applications where distributed and heterogeneous entities need to cooperate to achieve a common goal, or to concur for the control of shared resources. This paper proposes a declarative framework for developing multi-agent systems. A formal approach based on Logic Programming is proposed for the specification, implementation and testing of software prototypes. Specification of the PRS agent architecture is given as an example of application of our framework.

1 Introduction

Declarative languages, such as functional and logical languages, have mainly been used in the academic world. The use of imperative paradigms for the development of industrial software is usually motivated by reasons of efficiency. However, besides being reusable, declarative knowledge is more modular and flexible than imperative knowledge. It has better semantics, makes detecting and correcting contradictory knowledge easier, and provides abstraction of the (real) world in a natural way. Moreover, in this setting the use of meta-programming techniques provides a support for the integration of different kinds of knowledge. These features make the declarative paradigm a solution that is suitable for developing and verifying prototypes of complex applications, where a set of autonomous, intelligent and distributed entities cooperate and coordinate the exchange and integration of knowledge.

Agent-oriented technology [21, 16] faces the problem of modelling such kinds of applications. It is suitable for modelling entities which communicate (*social ability*), monitor the environment and react to events which occur in it (*reactivity*), are able to take the initiative whenever the situation requires so (*proactivity*) without human beings or other agents intervening (*autonomy*). Societies of such entities are called *Multi-Agent Systems (MAS)*. They take into account the *distribution* of the involved agents and the *integration* of heterogeneous software and data. These two issues are fundamental for the success of present software

systems and this is one of the reasons for the consensus that MAS have obtained in academia and industry. The combination of declarative and agent-oriented approaches has been studied over the last few years and it is very promising from both a theoretical and a practical point of view (see [9, 8, 10, 12]).

Unfortunately, at this time there is no evidence of a well-established engineering approach for building MAS-based applications. However, due to their inherent complexity, experimentation in this direction seems very important. This paper presents some features of **CaseLP** [11], an experimental, logic programming based, prototyping environment for MAS. In particular, we present a methodology that combines traditional software engineering approaches with considerations from the agent-oriented field. The approach exploits logic-based declarative languages for the specification, implementation and testing of the prototype. In our methodology the more formal and abstract specification of the MAS is given using the linear logic programming language \mathcal{E}_{hhf} [2], which provides constructs for concurrency and state-updating. Afterwards ACLPL, an extension of the logic programming language ECLiPSe with agent-oriented constructs, is used to build a software prototype closer to a final implementation of the MAS. The declarative nature of \mathcal{E}_{hhf} guides the translation into ACLPL, which, in turn, has a number of programming features making the resulting prototype more efficient and easier to integrate with other technologies.

In our framework we consider the possibility of building prototypes encompassing agents with different architectures. The designer will either choose a predefined architecture in a library that is part of the **CaseLP** environment or will develop a new one. In the former case, only the relevant data for a given agent will actually be provided by the user. As an example, in this paper we present the application of some steps of our methodology to the specification of the well known *Planning and Reasoning System (PRS)* agent architecture [6].

The paper is structured as follows: the next section presents our development framework, focusing on the prototype developing methodology. Section 3 describes the PRS architecture and how \mathcal{E}_{hhf} and ACLPL can be used to model this architecture following (part of) the methodology described in Section 2. Section 4 compares **CaseLP** with other frameworks for the specification and development of MAS, and concludes the paper with some considerations on future research work.

2 A Framework for MAS Development

The development of agent-based software can be seen as a traditional software engineering enterprise [20]. First, a specification in some suitable specification formalism is given, then it is refined into an executable piece of software. To assure correctness of the refinement process, the final concrete system has to be verified with respect to the initial specification. Formal methods play an important role in the development phase of agent-based systems at a high level, and when developing complex cooperating systems.

Since there is no generally accepted taxonomy for classifying agent-based applications, it is not yet clear which approaches are appropriate for specifying the different classes of agent-based systems and which methods can be exploited to transform these specifications into a final software product [3, 5]. In the last few years many approaches based on logical languages (e.g., temporal logic [15]) have been proposed as specification formalisms for agent-based technology. Logic programming languages can contribute to such research in that they provide easy-to-define, executable specifications. We propose a framework for the realization of MAS prototypes in which both the specification and implementation are carried out using declarative logical languages.

2.1 Prototype Realization

CaseLP (Complex Application Specification Environment Based on Logic Programming) [11] is a MAS-based framework for prototyping applications involving heterogeneous and distributed entities. More precisely, the framework includes a methodology that guides the application developer to an easy and rapid definition of the prototype through the iteration of a sequence of simple steps. CaseLP provides a set of tools that are used to achieve the aim of each step in the methodology. In particular, tools for specification of the MAS, tools for describing the behaviour of agents that make up the system by means of a simple rule-based logical language, tools for the integration of legacy systems and data, and simulation tools for animating the MAS execution are provided.

A CaseLP agent can be either a *logical agent*, which shows capabilities of complex reasoning, or an *interface agent*, which provides an interface between external modules and the agents in the system. The final prototype can be built by combining existing software tools and new components. This approach furthers integration and reuse, two issues that are highly relevant for the success of new technologies. All agents share some main components that are: an updatable set of facts defining the *state* of the agent, a fixed set of rules defining the *behaviour* of the agent, a *mail-box* for incoming messages and events, an *interpreter* for accessing external software (only for interface agents).

The language for high-level MAS specification is \mathcal{E}_{hhf} which has interesting capabilities for modelling concurrent and resource sensitive systems (see Section 3.1 below). The language for defining agent implementation is ACLPL, a Prolog-like language enriched with primitives for safe updates of the agent state (*assert_state(Fact)* and *retract_state(Fact)*) and for communication among agents in the MAS (*send(Message, Receiver)*, *asynch_receive(Message)* and *sync_receive(Message, Sender)*). *Message* follows the syntax of KQML [13], that we have chosen as the agent communication language. The update primitives operate in such a way that if the agent fails some activities a safe state is automatically restored. The primitive *asynch_receive(Message)* inspects the agent's mail-box and retrieves the first message contained in the mail-box, if any. This kind of reception is not blocking. On the other hand, *sync_receive(Message, Sender)* is the blocking reception primitive. The agent waits until a message coming from the agent *Sender* enters the mailbox.

CaseLP provides appropriate primitives for loading agents into the system and for associating an appropriate interpreter to interface agents. This creates a MAS that is ready for simulation which is performed by means of a *round-robin scheduler* interleaving the activation of all the agents. The simulation produces both on-line and off-line information about the agents' state changes and about the messages that have been exchanged. The CaseLP Visualizer provides a GUI for loading, initializing and tracing the agents' execution in a graphical, user-friendly manner.

Prototyping Methodology. The realization of a MAS-based software prototype can be performed according to the following steps:

1. **Static architectural description of the prototype.** The developer decides the static structure of the MAS. This step is further broken down to: (a) determining the *classes of agents* the application needs; (b) determining a set of *requested services* and a set of *provided services* for each class; (c) determining the set of *necessary instances* for each class; (d) defining the *interconnections* between the instances of agents, matching appropriately requested and provided services.
This phase defines the components, what they are able to do, what they require and which communication channels are needed to manage the services.
2. **Description of component interactions.** This step specifies how a service is provided/requested by means of a particular *conversation* (set of KQML messages with a specific order) between each pair of connected agents. Each conversation can be performed using different *communication models*, such as synchronous or asynchronous message passing.
3. **Architectural choice for each agent.** Each logical agent can be structured according to a particular agent architecture (e.g., only reactive or proactive agents may be needed). This step allows the developer to decide the most appropriate internal architecture for each agent in the system. For example, he/she decides a predefined architecture (such as the PRS that will be used as our guiding example in this paper) or must choose to build a specification of his/her own. Obviously, in the former case the steps involved in writing the specification will be much easier since dealing with the modelling of the internal mechanisms of the agents is not needed.
4. **High-level specification of the system.** In this step Linear Logic Programming comes into play and \mathcal{E}_{hhf} is used to build an executable specification of the system. Due to its peculiar properties, \mathcal{E}_{hhf} allows easy modellization of concurrency among agents, as well as updates of agents states. We can identify three different levels of modellization:
 - (a) specification of interactions among agents (external concurrency), abstracting from their architecture and taking into account the interaction model specified in step 2;
 - (b) specification of the (new) architectures chosen in step 3, i.e. modellization of the interactions between the internal components of the agents (internal concurrency);

- (c) specification of the agents' behaviour, i.e. how they operate to provide their services.

It is important to point out that the whole process listed in steps 1 through 4 may be repeated more than once, either because the testing phase (step 5) reveals some flaws in the initial choices, or because the developer decides to refine the specification. For example, in the first stage only specification 4a listed above could be given while 4b and 4c could be defined afterwards.

5. **Testing of the system.** This phase concerns testing the system in order to verify how closely the prototype corresponds to the desired requirements. Using a logical language like \mathcal{E}_{hhf} for this phase has numerous advantages and using the \mathcal{E}_{hhf} interpreter makes it possible to evaluate a goal step by step, following the evolution of a particular system in detail. It is possible to verify whether a particular computation may be carried out, or what is more important, that *every* computation starting from a given configuration leads to a final state which satisfies a given property. It might also be possible to employ standard techniques to prove properties of programs in the logic programming context. This is part of authors' future work.
6. **Implementation of the prototype.** At this point of the development process we have an abstract (hopefully correct) specification of the final application. This step transforms the \mathcal{E}_{hhf} specification into a prototype much closer to the final implementation (i.e., interfaces towards external software and data, message passing communication, etc.). Furthermore, performance and standardization reasons suggest using more efficient and widespread logical languages than \mathcal{E}_{hhf} for an actual implementation of MAS prototypes. In this step specifications 4a, 4b and 4c have to be translated into executable ACLPL code. 4a corresponds to various implementations of message exchange. 4b is translated into suitable data structures (obtaining different architecture parts) and into a meta-program that implements the control flow of the architecture. Finally, the architecture-dependent rules defining 4c are translated into ACLPL rules. The framework also allows the user to join all the agents that form the prototype into a unique executable specification, so that the system can be further tested. Obviously, in most cases only 4c has to be translated, if existing solutions are chosen for 4a and 4b.
7. **Execution of the obtained prototype.** The last step tests the implementation choices, checking whether the system behaves as expected. Any error or misbehaviour discovered in this step may imply a revision of the choices made in the previous steps.

Practical Application of CaseLP. CaseLP has been adopted in order to develop applications in different areas. Two applications were related to transportation and logistic problems. In particular one was developed in collaboration with FS (the Italian railway company) to solve train scheduling problems on the La Spezia – Milano route, and another one was developed with Elsag Bailey, an international company which provides service automation, to plan transportation of goods. Another application concerned the retrieval of medical information

contained in distributed databases. In this case CaseLP was successfully adopted for a reverse engineering process. Finally, the combination of agent-oriented and constraint logic programming techniques has been faced with CaseLP to solve the transaction management problem on a distributed database.

3 The PRS Architecture

The specification and implementation of the architecture of an agent are certainly among the most difficult phases of our development methodology. We will furnish CaseLP with a library of agent architectures from which the application developer will pick the desired model. In this section steps 4b and 6 of our methodology are applied to implement the PRS architecture.

The *Procedural Reasoning System (PRS)* [6] has obtained broad consensus among researchers in the multi-agent systems field. The model of practical reasoning that underpins PRS is BDI (Beliefs, Desires and Intentions) [18] which is operationalized in PRS agents by the notion of *plans*. Each agent has a plan library, which is a set of plans and represents the agent's procedural knowledge. Each plan consists of: a *trigger*, or invocation condition, which specifies the circumstances under which the plan should be considered; a *context*, or precondition, specifying the circumstances under which the execution of the plan may commence¹; a *maintenance condition*, which characterizes the circumstances that must remain true while the plan is executing, and a *body* defining a potentially complex course of actions which may consist of both *goals* and primitive *actions*. The agent's interpreter can be outlined by the following cycle [4]: (1) observe the world and the agent's internal state, and update the *event queue* consequently; (2) generate possible new *desires* (tasks) by finding plans whose trigger event matches an event in the event queue; (3) select one from this set of matching plans for execution; (4) push the selected plan onto an existing or new *intention stack*, according to whether or not the event is a (sub)goal; (5) select an intention stack, take the topmost plan and execute the next step of this current plan: if the step is an action, perform it, otherwise, if it is a subgoal, post it on the event queue.

On the basis of work in [4], a specification of dMARS (an implementation of PRS) using the Z specification language [19], we will show how this architecture can be modelled using \mathcal{E}_{hhf} and CaseLP. First we briefly introduce some basic features of linear logic programming [2, 14].

3.1 Executable Specifications in \mathcal{E}_{hhf}

The linear logic language \mathcal{E}_{hhf} [2] is an executable language for modelling concurrent and resource sensitive systems based on the general purpose logical specification language Forum [14]. \mathcal{E}_{hhf} is a multiset-based logic combining features of extensions of logic programming languages like λ Prolog, e.g. goals with implication and universal quantification, with the notion of *formulas as resources* at the

¹ Note that a triggered plan can start its execution only if its context is satisfied

basis of linear logic. Below we will informally describe the operational semantics of \mathcal{E}_{hhf} -programs. Specifically, \mathcal{E}_{hhf} -programs are a collection of multi-conclusion clauses of the form:

$$A_1 \wp \dots \wp A_n \multimap Goal,$$

where the A_i 's are atomic formulas, and the linear disjunction $A_1 \wp \dots \wp A_n$ corresponds to the head of the clause. Furthermore, $A \multimap B$ (i.e., $B \multimap A$) is a linear implication. The main peculiarity of such clauses is that they *consume* the resources (formulas) they need in order to be applied in a resolution step.

Formally, given a multiset of atomic formulas (the state of the computation) Ω_0 , a resolution step $\Omega_0 \rightarrow \Omega_1$ can be performed by applying an instance $A_1 \wp \dots \wp A_n \multimap G$ of a clause in the program P , whenever the multiset Θ consisting of the atoms A_1, \dots, A_n is contained in Ω_0 . Ω_1 is then obtained by removing Θ from Ω_0 and by adding G to the resulting multiset. In the interpreter, instantiation is replaced by unification. At this point, since G may be a complex formula, the search rules (i.e., the logical rules of the connectives occurring in G) must be exhaustively applied in order to proceed. Such a derivation, which corresponds to a branch of the proof tree of the multiset Ω , can be used to model the evolution of a PRS agent. Ω represents the current global state, e.g., *beliefs*, *intentions* and the *event queue*, whereas P describes the possible *plans* that can be triggered at any point during a computation. \mathcal{E}_{hhf} provides a way to “guard” the application of a given clause. In the following extended type of clauses

$$G_1 \& \dots \& G_m \Rightarrow (A_1 \wp \dots \wp A_n \multimap Goal),$$

the goal-formulas G_i 's must be solved (i.e., executed in P) in order for the clause to be triggered. New components can be added to the current state by using goal-formulas of the form $G_1 \wp G_2$. In fact, the goal $G_1 \wp G_2, \Delta$ simply reduces to G_1, G_2, Δ . Conditions over the current state can be tested by using goal-formulas of the form $G_1 \& G_2$. In fact, the goal $G_1 \& G_2, \Delta$ reduces to G_1, Δ and G_2, Δ . Thus, one of the two copies of the state can be consumed to verify a contextual condition. Universal quantification in a goal-formula $\forall x.G$ can be used to create a new identifier t which must be local to the derivation tree of the subgoal $G[t/x]$. Finally, the constant \top succeeds in any context and the constant \perp is simply removed from the current goal. Such a description can be used to observe the evolution of an agent or, by using backward analysis, to detect violations of the requirements of the specifications.

3.2 LLP Specification of a PRS Agent

In this section we explain in detail how to specify the salient aspects of the PRS architecture using \mathcal{E}_{hhf} .

Beliefs, Goals, Actions and Plans. *Beliefs* can be modelled by ground facts of the form *belief(Fact)*. The set of beliefs is maintained in the current state Ω . *Goals* are terms of the form *achieve(Fact)* and *query(Fact)*. An *achieve goal* commits the agent to a sequence of actions, whose execution must make the goal true. A *query goal* implies a test on the agent's beliefs to know whether the

goal is true or not. *Internal actions*, represented by the terms *assert(Fact)* and *retract(Fact)*, update the agent beliefs. *External actions* (e.g. sending messages) are denoted by generic terms. *Plans* basically consist of sequences of actions and (sub)goals. They are represented as facts in the following form

$$\text{plan}(\text{Trigger}, \text{Context}, \text{Body}, \text{Maintenance}, \text{SuccActs}),$$

where *Trigger* is a trigger linked to a particular event, *Context* is a condition that has to be true to start the plan, *Body* is a sequence of actions and (sub)goals $B_1 :: \dots :: B_n$, *Maintenance* is a condition that has to be true while the plan is executed, *SuccActs* is a sequence of actions to be executed if the plan succeeds. Thus, the *plan library* of the agent is described as a collection of facts. On the contrary to [4] we do not take into consideration *failure actions* whose aim is generally to rollback to a safe state in case the plan fails. This can be obtained without effort in an LP setting by exploiting backtracking.

Triggers and events. Triggers are raised by events and their aim is to activate plans. We can distinguish between *external triggers*, that force an agent to adopt a new intention, and *internal triggers*, that cause an agent to add a new plan to an already existing intention. External triggers are: *addbelief(Fact)* and *delbelief(Fact)*, linked respectively to events denoting acquisition and removal of a belief, and *goal(G)* denoting acquisition of a new goal. Instead, there is only one internal trigger, i.e., *subgoal(G)*, denoting a (sub)goal call in the executing plan. Events are terms of the form *event(Trig, Id)*, where *Trig* can assume one of the forms above, and *Id* is a (possibly undefined) identifier of the plan instance causing the event.

Event queues. An event queue is associated with each PRS agent. It contains external and internal events and is represented by a term of the form

$$\text{event_queue}([E_1 \mid \dots \mid E_n]),$$

where $E_1 \dots E_n$ are events. Events linked to external triggers are always inserted at the end of the event queue, whereas events linked to internal triggers are inserted at the top. Since events are taken from the top of the event queue, this policy gives priority to events generated by an attempt to achieve a (sub)goal.

Plan instances. Plan instances represent plans to execute. A new plan instance is created and inserted into an *intention stack* as soon as its trigger has been activated and its context is satisfiable. A plan instance contains an instantiated copy of the original plan from which it derives together with and the information about whether the plan is *active* or *suspended*. Differently from [4], we use shared variables and unification to inherit knowledge from the original plan. A unique identifier is associated to each plan instance. Thus, plan instances have the form

$$\text{plan_inst}(\text{Id}, \text{Body}, \text{Maintenance}, \text{Active}, \text{SuccActs}).$$

Intention stacks. An intention stack contains the currently active plan and a set of suspended plans. An internal trigger *subgoal(G)* can generate a new

plan instance that is pushed onto the same intention stack as the plan containing the (sub)goal whose execution has activated the trigger. An external trigger produces a new plan instance that is stored in a new intention stack. In our representation an intention stack is a term of the form

$$int_stack([P_1 \mid \dots \mid P_n]),$$

where $P_1 \dots P_n$ are plan instances. The internal behaviour of a PRS agent can be described by means of the \mathcal{E}_{hhf} rules listed below. For the sake of this example we do not explicitly give rules for *perception*, i.e., rules for simulating interactions with the external world. As in [4], the idea is to start the simulation of the agent from an initial queue of events and from an initial intention stack. During execution, perception rules can be used to build appropriate events according to information produced by some perception devices (for example environment sensors or inter-agent communication devices) and to (non deterministically) post them into the agent event queue. Other than the rules listed below, we consider a program P containing the plan library and the information to distinguish external and internal triggers and actions, of the form *external*(*Trigger*), *external*(*Action*), etc.

Plan triggering. Plan triggering rules handle creation of new plan instances and intention stacks, according to the raised trigger. If an external trigger has to be handled, a new intention stack is created and a new plan instance is pushed onto it. This can be formalized by the following rule:

$$plan(T, C, B, M, SA) \ \& \ external(T) \Rightarrow (event_queue([event(T, -) \mid L]) \ \circ - \\ verify(C) \ \& \ (\forall Id. int_stack([plan_inst(Id, B, M, act, SA)])) \ \wp \ event_queue(L))).$$

The conditions *plan*(...) and *external*(...) must be fulfilled by P (i.e. must be unified with facts in P), whereas the goal-formula *verify*(...) & ($\forall Id. \dots$) allows us to test the contextual condition over a copy of the current state (for sake of brevity *verify* is not specified here). New identifiers for the plan instances are created by using universal quantification. Finally, note that the modification of the event queue is defined by *consuming* the current one (the head of the clause) and creating a new copy (in the body).

A similar clause formalizes what to do if the trigger is internal. The main difference is that the intention stack, whose top plan has the same identifier as the first event in the event queue, is *consumed*. After verifying the context, the intention stack is rewritten adding a new instance of the plan whose trigger matches the trigger of the topmost event in the event queue.

Plan execution. A plan is executed when the event queue of the agent is empty. An intention stack such that its top plan instance is active is non deterministically chosen and the first action of the plan instance is executed (if the maintenance condition is verified). We have developed rules for each possible plan component: external and internal actions, query goals and achieve goals. For sake of brevity we only report the most significant among them.

In case an external action has to be executed, the following rule is applied:

$$\begin{aligned} \text{external}(A) \Rightarrow & (\text{event_queue}([]) \text{ } \wp \text{ } \text{int_stack}([\text{plan_inst}(Id, A :: B, M, \text{act}, SA)|L]) \\ \circ - & \text{verify}(M) \ \& \ (\text{event_queue}([]) \text{ } \wp \text{ } \text{execute}(A) \text{ } \wp \\ & \text{int_stack}([\text{plan_inst}(Id, B, M, \text{act}, SA)|L])). \end{aligned}$$

The call to $\text{execute}(A)$ activates the agent output devices, for example effectors on the environment, or the inter-agent communication for sending a message to another agent.

In case of internal actions the following rules are applied; if the top component of a plan is an *assert* action, the corresponding belief is added to the database (if it is not present):

$$\begin{aligned} \text{event_queue}([]) \text{ } \wp \text{ } \text{int_stack}([\text{plan_inst}(Id, \text{assert}(F) :: B, M, \text{act}, SA)|L]) \circ - \\ \text{verify}(M) \ \& \ \text{not_believed}(F) \ \& \ (\text{belief}(F) \text{ } \wp \\ \text{event_queue}([\text{event}(\text{addbelief}(F), Id)]) \text{ } \wp \\ \text{int_stack}([\text{plan_inst}(Id, B, M, \text{active}, SA)|L])). \end{aligned}$$

Similarly, if the top component of a plan is a *retract* action, then the corresponding belief (if present) is removed by *consuming* it.

If the plan component is an *achieve* goal, and the corresponding belief is in the database, then the agent can proceed:

$$\begin{aligned} \text{event_queue}([]) \text{ } \wp \text{ } \text{belief}(F) \text{ } \wp \\ \text{int_stack}([\text{plan_inst}(Id, \text{achieve}(F) :: B, M, \text{act}, SA)|L]) \circ - \text{verify}(M) \ \& \\ (\text{event_queue}([]) \text{ } \wp \text{ } \text{belief}(F) \text{ } \wp \text{ } \text{int_stack}([\text{plan_inst}(Id, B, M, \text{act}, SA)|L])). \end{aligned}$$

If the previous rule cannot be applied, the current plan is suspended and a new event containing a *subgoal* trigger and a reference to the current plan is created:

$$\begin{aligned} \text{event_queue}([]) \text{ } \wp \text{ } \text{int_stack}([\text{plan_inst}(Id, \text{achieve}(F) :: B, M, \text{act}, SA)|L]) \circ - \\ \text{verify}(M) \ \& \ \text{not_believed}(F) \ \& \ (\text{event_queue}([\text{event}(\text{subgoal}(F), Id)]) \text{ } \wp \\ \text{int_stack}([\text{plan_inst}(Id, B, M, \text{susp}, S)|L])). \end{aligned}$$

If the plan component is a *query* goal and F is a belief of the agent, then the query succeeds by using a rule similar to the first *achieve* rule (the plan instance at the top of the intention stack is a $\text{query}(F)$ instead of an *achieve*(F)).

Plan termination and resuming. If a plan has been completed successfully, the *SuccActs* (sequence of internal actions) has to be executed. Furthermore, awakening a previously suspended plan may be necessary. The following rules capture execution of *SuccActs*:

$$\begin{aligned} \text{event_queue}([]) \text{ } \wp \text{ } \text{int_stack}([\text{plan_inst}(Id, [], \neg, \text{act}, \text{assert}(F) :: SA)|L]) \circ - \\ \text{not_believed}(F) \ \& \ (\text{event_queue}([]) \text{ } \wp \text{ } \text{belief}(F) \text{ } \wp \\ \text{int_stack}([\text{plan_inst}(Id, [], \neg, \text{act}, SA)|L])). \end{aligned}$$

$$\begin{aligned} \text{event_queue}([]) \text{ } \wp \text{ } \text{belief}(F) \text{ } \wp \text{ } \text{int_stack}([\text{plan_inst}(Id, [], \neg, \text{act}, \text{retract}(F) :: SA)|L]) \\ \circ - \text{event_queue}([]) \text{ } \wp \text{ } \text{int_stack}([\text{plan_inst}(Id, [], \neg, \text{act}, SA)|L])). \end{aligned}$$

When the execution of *SuccActs* has been completed, the plan instance can be popped away from the intention stack, and the (suspended) lower plan instance has to be awakened:

$$\begin{aligned} & \text{event_queue}(\square) \text{ } \mathfrak{?} \text{ int_stack}([\text{plan_inst}(_, \square, _, _, \square), \text{plan_inst}(Id, B, M, \text{susp}, SA)|L]) \\ & \quad \multimap \text{event_queue}(\square) \text{ } \mathfrak{?} \text{ int_stack}([\text{plan_inst}(Id, B, M, \text{act}, SA)|L]). \end{aligned}$$

When an intention stack becomes empty, it is removed using the rule

$$\text{int_stack}(\square) \multimap \perp.$$

As mentioned before, such a specification can be directly executed in a logic programming language by setting an initial set of events and of intentions. The next step in the **CaseLP** methodology is to refine the specification in order to get closer to a real implementation.

3.3 LP Prototyping of a PRS Agent

The refinement of the \mathcal{E}_{hhf} specification into a more concrete prototype aims mainly at the realization of a software product based on a more established technology and in which modules written in different languages can be integrated. Moreover, we consider the possibility of building prototypes encompassing agents with different architectures and we aim at providing the **CaseLP** environment with a library of meta-interpreters, each of which reproduces a particular kind of management of the agent behaviour. After being abstractly specified with \mathcal{E}_{hhf} , these meta-interpreters will (automatically) be translated into the more concrete ACLPL language. As a first step towards this library, we present an interpreter for the PRS architecture which is directly derived from the previously defined \mathcal{E}_{hhf} specification.

An interpreter for the PRS architecture. Mapping the \mathcal{E}_{hhf} specification of PRS into a **CaseLP** agent does not require a great effort. In fact, the \mathcal{E}_{hhf} data structure previously defined finds a direct mapping into ACLPL terms. Furthermore, representation of *beliefs* and *intentions* is stored in the agent *state*, the *event queue* corresponds to the agent *mail-box*, and *plans* are memorized in the agent *behaviour* as facts.

We can note that the \mathcal{E}_{hhf} clauses, used for specifying the PRS interpreter, assume the general form

$$G_1 \& \dots \& G_m \Rightarrow (A_1 \mathfrak{?} \dots \mathfrak{?} A_n \multimap G_{m+1} \& \dots \& G_{m+k} \& A_{n+1} \mathfrak{?} \dots \mathfrak{?} A_{n+h}),$$

where all the formulas $G_1, \dots, G_{m+k}, A_1, \dots, A_{n+h}$ are atomic. In order to translate a multi-conclusion guarded clause C of such a form, we can introduce an auxiliary predicate p_C , defined as

$$\begin{aligned} p_C \text{ :- } & \text{retract_state}(A_1), \dots, \text{retract_state}(A_n), G_1, \dots, G_{m+k}, \\ & \text{assert_state}(A_{n+1}), \dots, \text{assert_state}(A_{n+h}). \end{aligned}$$

where *retract_state* and *assert_state* are the state update predicates previously described. The execution of $\text{retract_state}(A_1), \dots, \text{retract_state}(A_n)$ consumes the atomic formulas A_1, \dots, A_n , the proof of G_1, \dots, G_{m+k} tests both the clause guard and the condition over the current state, and finally the execution of

assert_state(A_{n+1}), ..., *assert_state*(A_{n+h}) adds new information to the state itself.

Applying this transformation to every \mathcal{E}_{hhf} clause, we can obtain a corresponding set of **CaseLP** clauses. Such a set can be partitioned by grouping together clauses regarding the four main activities performed by a PRS agent that are *perception*, *plan_triggering*, *plan_execution* and *action_execution*. The top-level behaviour of a simple version PRS interpreter can be given by the rule that starts the four main activities, with a priority that reflects their order in the body of the clause.

prs_interpreter:- *perception*; *plan_triggering*; *plan_execution*; *action_execution*.

Obviously, more sophisticated strategies can be implemented, so that the execution of the four activities may take additional information into account.

Each activity can be defined by the clauses belonging to the corresponding activity group, for example

plan_triggering:- p_{C_1} ; ... ; p_{C_s} .

where p_{C_1}, \dots, p_{C_s} are the clauses of the plan triggering group.

We do not present the complete code for the PRS interpreter, but we only give (part of the) rules for handling internal triggers and external actions. The rule for handling an internal trigger (belonging to the definition of *plan_triggering*, together with rules for external triggers) is

plan_triggering:- *retract_state*(*mail_box*([*event*($T, Id1$)| L]),
retract_state(*int_stack*([*plan_instance*($Id1, B_1, M_1, act, SA_1$)| L_1)]),
plan(T, C, B, M, SA), *internal*(T), *verify*(C), *get_new_id*(Id),
assert_state(*mail_box*([L]),
assert_state(*int_stack*([*plan_instance*(Id, B, M, act, SA),
plan_instance($Id1, B_1, M_1, act, SA_1$)| L_1)]).

In this clause the call of a p_{C_i} has been substituted by its definition and the goal *get_new_id*(Id) creates a new identifier.

To execute a plan step involving an external action the interpreter uses the rule

plan_execution:-
retract_state(*int_stack*([*plan_instance*($Id, [A|Body], Maint, active, SA$)| L)]),
retract_state(*mail_box*([])), *external*(A), *verify*($Maint$),
assert_state(*int_stack*([*plan_instance*($Id, Body, Maint, active, SA$)| L)]),
assert_state(*mail_box*([])), *assert_state*(*execute*(A)).

The rule for executing an external action is

action_execution:- *retract_state*(*execute*(*send*($Message, Receiver$))),
send($Message, Receiver$).

where the verification of goal *send*($Message, Receiver$) has the side effect of performing the message send. Note that in **CaseLP** the only allowed external actions concern sending messages, so this specialized rule can be used.

The PRS interpreter is started whenever the agent is awakened by the system scheduler. This is achieved by calling the hook predicate *activate* defined as

activate:- *prs_interpreter*.

After the PRS agent has executed one of the four main activities, control returns to the system scheduler so that another agent can be activated.

The above translation of the \mathcal{E}_{hhf} specification into the concrete logical language ACLPL could be further refined into a more efficient one by exploiting advanced features of the ECLiPSe system.

4 Comparison and Future Work

In our paper we have described **CaseLP** as a tool which adopts software engineering techniques to develop multi-agent system prototypes. Logic is used both as the prototype specification and implementation language. How does **CaseLP** compare with other general purpose tools for MAS development and testing?

MIX [7] has been conceived as a distributed framework for the cooperation of multiple heterogeneous agents. Its basic components are the agents and the network through which they interact. A *yellow page* agent offers facilities for receiving information on the dynamic changes of the network environment.

The **Open Agent Architecture** (OAA) [12] provides software services through the cooperative efforts of distributed collections of autonomous agents. There are different categories of agents recognized in the framework. The *facilitator* is a server agent responsible for coordinating agent communications and cooperative problem-solving. *Application agents* are “specialists” that provide a collection of services (eventually based on legacy applications). *Meta-agents* help the facilitator agent during its multi-agent coordination phase, and the *user interface agent* provides an interface to the user.

ZEUS [17] is an advanced development tool-kit for constructing collaborative agent applications. It defines a multi-agent system design methodology, supports the methodology with an environment for capturing user specification of agents and automatically generates the executable source code for the user-defined agents. A **ZEUS** agent is composed of a definition layer, an organization layer and a coordination layer. **ZEUS** also provides predefined yellow and white pages agents.

ZEUS appears to be the most similar to **CaseLP**. The proposed methodology is given as a set of questions, whose answers the MAS developer uses to define the agent in terms of its role, the services it provides and the relationships with other agents. Even though it is less formalized, it has the same aim as the **CaseLP** methodology. A feature which characterizes all the presented approaches except for **CaseLP** is the presence of a predefined agent which helps discover which agent provides which services (yellow and white pages in **ZEUS**, yellow pages in **MIX** and facilitator in **OAA**). It would not be difficult to provide **CaseLP** with such an agent but, at the moment, it is up to the user to implement it, if necessary. Integration of legacy software is dealt with in **MIX**, **OAA** and **ZEUS** by adopting a *wrapping* approach that “agentifies” existing software. On the contrary **CaseLP** adopts an *interpretation* approach.

As an example of specification of MAS architecture, we have presented a possible realization of the PRS one. We think the approach we followed for

the PRS specification may have some advantages over other ones present in the literature. In [4] a specification for PRS is given using Z, a specification language based on set theory, first-order logic, and the notions of state space and transformations between states. We think that Linear Logic may be a more natural candidate for the specification of systems where the notion of state, transformations and resources are involved. The main advantage of \mathcal{E}_{hhf} with respect to Z is its being directly *executable*. It also supports specification at different levels of abstraction like Z, but its higher-order extensions (Z is first-order) greatly facilitate meta-programming.

Another formal specification for PRS is presented in [15]. It is based on a particular temporal logic language, **Concurrent MetateM**, which supports the specification of concurrent and distributed entities. Goals, beliefs and plans are represented by means of formulas in temporal logic, whose execution is committed to a run-time execution algorithm. Verification of the specifications is rather direct even though at the moment a small, fast interpreter for **Concurrent MetateM** is not available. On the contrary, the language we propose is based on a linear extension of classical logic languages such as Prolog. Its execution mechanism is goal directed and is based on clause resolution and unification as is usual in Logic Programming. \mathcal{E}_{hhf} , like **Concurrent MetateM**, easily supports broadcast message-passing and can simulate meta-level activity by means of its higher-order features.

An experimental interpreter for \mathcal{E}_{hhf} has been developed by the authors (see <ftp://ftp.disi.unige.it/pub/person/BozzanoM/Terzo>). The development of a more efficient interpreter for the language is part of our future work. Furthermore, we also plan to investigate the possibility of extending standard Logic Programming techniques for software verification and validation to the Linear Logic context. Among these, it would be worth considering *symbolic model checking*, and possibly techniques based on *partial evaluation* and *abstract interpretation*. As far as **CaseLP** is concerned, we plan to extend the system to allowing for a *real* distribution of the agents on a network, so that prototypes will be closer to final implementation. Finally, as described in [1], we intend to integrate different research experiences based on Logic Programming, including **CaseLP**, into a common joint project which will lead to the development of the general open framework **ARPEGGIO** (Agent based Rapid Prototyping Environment Good for Global Information Organization), for the specification, rapid prototyping and engineering of agent-based software. The **ARPEGGIO** framework will also include work on integration of multiple data sources and reasoning systems being carried out by the Department of Computer Science at the University of Maryland (USA), as well as work being done on animation of specifications at the Department of Computer Science at the University of Melbourne (Australia).

References

- [1] P. Dart, E. Kazmierczak, M. Martelli, V. Mascardi, L. Sterling, V.S. Subrahmanian, and F. Zini. Combining Logical Agents with Rapid Prototyping for Engineering Distributed Applications. Submitted to FASE'99.
- [2] G. Delzanno. *Logic & Object-Oriented Programming in Linear Logic*. PhD thesis, Università di Pisa, Dipartimento di Informatica, 1997.
- [3] M. d'Inverno, M. Fisher, A. Lomuscio, M. Luck, M. de Rijke, M. Ryan, and M. Wooldridge. Formalisms for Multi-Agent Systems. *The Knowledge Engineering Review*, 12(3), 1997.
- [4] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge. A Formal Specification of dMARS. In *Intelligent Agents IV*. Springer-Verlag, 1997. LNAI 1365.
- [5] M. Fisher, J. Mueller, M. Schroeder, G. Staniford, and G. Wagner. Methodological Foundations for Agent-Based Systems. *The Knowledge Engineering Review*, 12(3), 1997.
- [6] M. Georgeff and A. Lansky. Reactive Reasoning and Planning. In *Proc. of the Sixth National Conference on Artificial Intelligence (AAAI-87)*, Seattle, WA, 1987.
- [7] C. A. Iglesias, J. C. Gonz  les, and J. R. Velasco. MIX: A General Purpose Multiagent Architecture. In *Intelligent Agents II*. Springer-Verlag, 1995. LNAI 1037.
- [8] R. Kowalsky and F. Sadri. Towards a Unified Agent Architecture that Combines Rationality with Reactivity. In *Proc. of International Workshop on Logic in Databases*, San Miniato, Italy, 1996. Springer-Verlag.
- [9] Y. Lesperance, H. Levesque, F. Lin, D. Marcu, R. Reiter, and R. B. Scherl. Foundations of a Logical Approach to Agent Programming. In *Intelligent Agents II*. Springer-Verlag, 1995. LNAI 1037.
- [10] S. W. Locke, L. Sterling, L. Sonenberg, and H. Kim. ARIS: A Shell for Information Agents that Exploit Web Site Structure. In *Proc. of PAAM'98*, London, UK, 1998.
- [11] M. Martelli, V. Mascardi, and F. Zini. Towards Multi-Agent Software Prototyping. In *Proc. of PAAM'98*, London, UK, 1998.
- [12] D. L. Martin, A. J. Cheyer, and D. B. Moran. Building Distributed Software Systems with the Open Agent Architecture. In *Proc. of PAAM'98*, London, UK, 1998.
- [13] J. Mayfield, Y. Labrou, and T. Finin. Evaluation of KQML as an Agent Communication Language. In *Intelligent Agents II*. Springer-Verlag, 1995. LNAI 1037.
- [14] D. Miller. Forum: A Multiple-Conclusion Specification Logic. *Theoretical Computer Science*, 165(1), 1996.
- [15] M. Mulder, J. Treur, and M. Fisher. Agent Modelling in METATEM and DESIRE. In *Intelligent Agents IV*. Springer-Verlag, 1997. LNAI 1365.
- [16] D. T. Ndumu and H. S. Nwana. Research and development challenges for agent-based systems. *IEEE Proc. of Software Engineering*, 144(1), 1997.
- [17] H. S. Nwana, D. T. Ndumu, and L. C. Lee. ZEUS: An Advanced Tool-Kit for Engineering Distributed Multi-Agent Systems. In *Proc. of PAAM'98*, London, UK, 1998.
- [18] A. S. Rao and M. Georgeff. BDI Agents: from Theory to Practice. In *Proc. of ICMAS'95*, San Francisco, CA, 1995.
- [19] M. Spivey. *The Z Notation (second edition)*. Prentice Hall International, 1992.
- [20] M. Wooldridge. Agent-based Software Engineering. *IEEE Proc. of Software Engineering*, 144(1), 1997.
- [21] M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 10(2), 1995.

From Functional Animation to Sprite-Based Display

Conal Elliott

Microsoft Research

<http://research.microsoft.com/~conal>

Abstract. Functional animation encourages a highly modular programming style, by supplying a set of arbitrarily composable functions for building up animations. In contrast, libraries for sprite-based display impose rigid structure, in order to allow acceleration by hardware and low level software. This paper presents a method to bridge the gap between functional specification and stateful, sprite-based presentation of animation. The method's correctness is proved informally by derivation from a simple non-effective specification, exploiting algebraic properties of the animation data types that are made explicit in the functional approach. We have implemented this method in the *Fran* system, which is freely available.

1 Introduction

The functional approach to animation offers considerable flexibility and modularity [1, 7]. Animations are first-class values—elements of a data type consisting of a set of constants and combining operators. The data type allows great flexibility in composing these basic building blocks into either directly useful or attractive animations, or new building blocks, parameterized as desired. Moreover, animation is a polymorphic notion, applying to 2D images, 3D geometry, and constituent types like colors, points, vectors, numbers, booleans, etc. Consequently, there is not just one animation type, but a collection of types and type constructors. In a well-designed set of data types, the type system imposes just enough discipline to rule out nonsensical compositions (such as rotating by the angle “purple”), without inhibiting the author’s creativity. In this way, the data types are designed to serve the needs of the author (and readers) of an animation.

Lower level graphics presentation libraries are designed not for convenience of a program’s author or readers, but rather for efficient execution on anticipated hardware. Programs written directly on top of these libraries must adapt to relatively inflexible representations and tend to be relatively non-modular.

To illustrate the convenience of functional animation, consider the animation in Figure 1. The function `repSpinner` repeatedly transforms an animation `im`,

¹ <ftp://ftp.research.microsoft.com/pub/tr/tr-98-28/animations.htm> contains running versions of this example and a few others. Those versions were recorded at 20 fps (frames per second), but run in *Fran* at 60 fps.

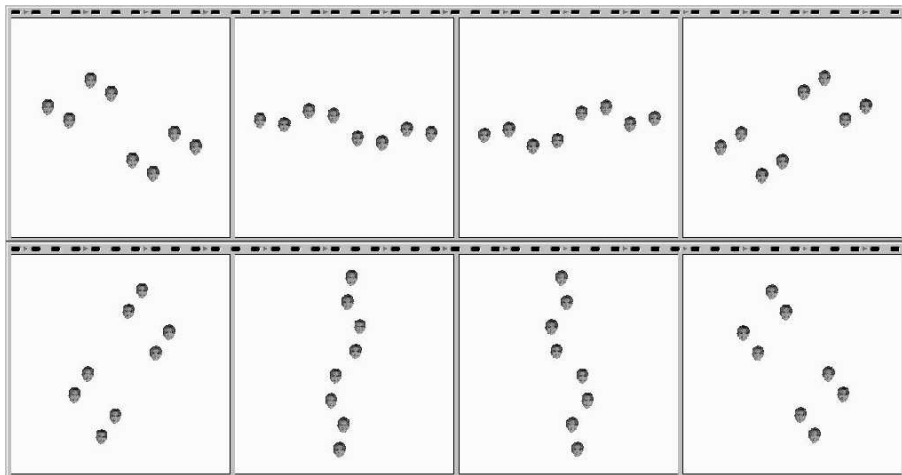


Fig. 1. repSpinner (-1) 0.5 charlotte 3

stretching `im` by `r`, speeding it up by `s`, and putting two copies into circular orbit. (`iterate f x` produces the infinite list `[x, f x, f (f x), ...]`, and `l!!n` extracts the `n`-th element of the list `l`. Their use together here results in applying `spinner` to `im`, `n` times.)

```
repSpinner :: RealB -> RealB -> ImageB -> Int -> ImageB
repSpinner r s im n = iterate spinner im !! n
  where
    spinner im = orbit 'over' later 1 orbit
      where
        orbit = move path (faster r (stretch s im))
        path  = vector2Polar 0.5 (pi*time)
```

In this example, time- and space-transforming functions (`faster` and `stretch`) are applied to overlays of animations that contain more time- and space-transformed animations (when $n > 1$), as shown in Figure 2(a). In contrast, sprite-based subsystems impose rigid structuring on an animation. It must be a sequence of implicitly overlaid “sprites”, where each sprite is the result of applying possibly time-varying motion and scaling to a “flip-book” of bitmaps, as in Figure 2(b). (In our example, the flip-book has only one page.) It is tedious and error-prone for a programmer or artist to work with such restrictions, when the animated objects of interest have natural hierarchical structure, as in `repSpinner`. The required sprite motion and scaling vectors (`moti` and `sci`) are quite complex, because each one is affected by a number of space- and time-transforms above them in the original hierarchy.

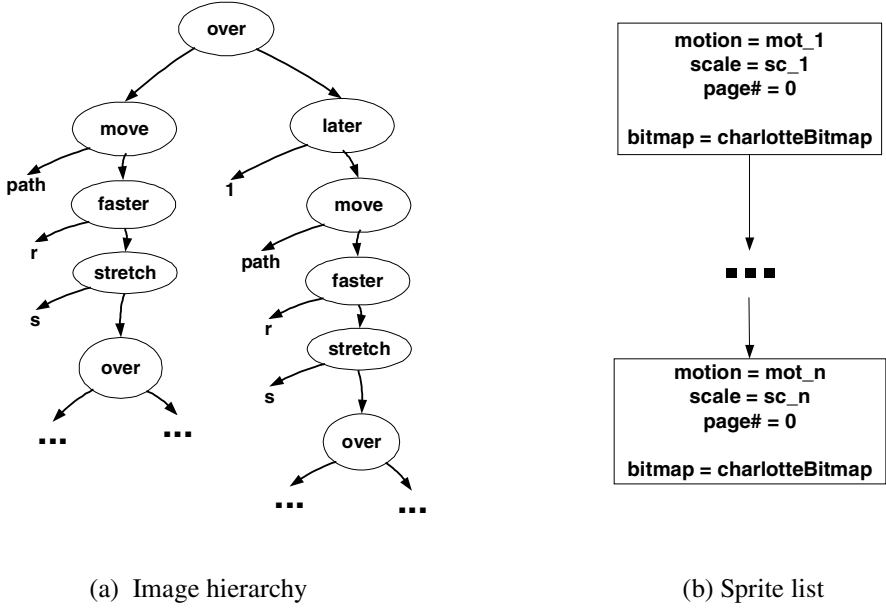


Fig. 2. Image hierarchies vs. sprite lists

This paper presents an algorithm to convert functional animations into a “sprite normal form” amenable to a fast sprite-based presentation library. As such, it bridges the gap between the high level flexibility desired for specification of animations and the low level regularity imposed by the presentation library. This algorithm is used in the implementation of *Fran* (“Functional Reactive Animation”) [7, 3, 4], which is written in Haskell [9, 8] and runs in conjunction with a fast “temporal sprite engine,” implemented in C++ for use with *Fran* and running in a separate thread. The sprite engine, constructed for use with *Fran* and described in this paper, manages a list of sprites. While some sprites are rendered on the fly, the sprite engine also supports fast flip-book animation. Typically, *Fran* updates information maintained by the sprite engine ten times per second, while the sprite engine interpolates motion, scaling, and indexing, and updates the screen, all at sixty times per second.

In order to transform flexibly specified animations into the form required by the sprite engine, we exploit several algebraic properties. In addition to time- and space-transformation, *Fran*’s animation algebra supports cropping, conditionals and reactivity, each requiring special care for conversion to sprite normal form.

Fran grew out of an earlier design called ActiveVRML [2], which was a ML-like language extended with reactive behaviors and multimedia. In contrast, *Fran* is a Haskell library. This “embedding” approach, discussed in detail elsewhere [3], has allowed considerable leverage from the design and implementation of the

```

emptyImage  :: ImageB          -- transparent everywhere
solidImage  :: ImageB          -- solid color image
flipImage   :: HFlipBook -> RealB -> ImageB -- flipbook-based
renderImage :: Renderer -> ImageB    -- text, 2D & 3D geometry
soundImage  :: SoundB -> ImageB      -- embedded sound
over        :: ImageB -> ImageB -> ImageB -- overlay
withColor   :: ColorB -> ImageB -> ImageB -- colored image
crop        :: RectB  -> ImageB -> ImageB -- cropped image

(*%)        :: Transform2B -> ImageB -> ImageB -- apply 2D transform
untilB      :: ImageB -> Event ImageB -> ImageB -- reactive
condB       :: BoolB -> ImageB -> ImageB -> ImageB -- conditional
timeTransform :: ImageB -> TimeB -> ImageB

```

Fig. 3. Abstract interface to the `ImageB` type

host language Haskell. Fran is freely available, including the sprite engine, in full source code at

<http://research.microsoft.com/~conal/fran>

2 Animation Data Types

Fran’s animation data types have been described elsewhere (e.g., [7], and for reference [11]). The animation-level types other than animated images, sounds, and 3D geometry come from applying the `Behavior` type constructor to these static types, and lifting operations on the static types to operations on behaviors. Behaviors are functions of continuous time. As a convention, the behavior types have synonyms made by adding a “B” to the static type name, e.g., “`Transform2B`” is a synonym for “`Behavior Transform2`”.

The type `ImageB` represents image animations that are spatially and temporally continuous and infinite. The primitives used in the construction of `ImageB` values are shown in Figure 3. (The type `Event a`, for an arbitrary type `a`, represents a stream of time-stamped values of type `a`, and is the basis of reactivity in behaviors.)

The `renderImage` function is used for all of the various kind of synthetic images, based on text, 2D geometry, 3D geometry, etc. The type `Renderer` maps time-varying cropping rectangle, color, scale factor, rotation angle, and time transform to a time-varying bitmap:

```

type Renderer = RectB -> Maybe ColorB -> RealB -> RealB
              -> TimeB -> SurfaceULB

-- Bitmap plus upper-left corner location
data SurfaceUL = SurfaceUL HDDSurface Point2
type SurfaceULB = Behavior SurfaceUL

```

```

(crop rectB          $
 mbWithColor mbColorB $
 stretch scaleB     $
 turn    angleB      $
 ('timeTransform' tt) $
 RenderImage renderer )
==
surfaceIm (renderer rectB mbColorB scaleB angleB tt)

-- Possibly apply a color
mbWithColor :: Maybe ColorB -> ImageB -> ImageB
mbWithColor Nothing imB = imB
mbWithColor (Just c) imB = withColor c imB

-- The ImageB contained on a discrete image (for specification).
surfaceIm :: SurfaceULB -> ImageB

```

Fig. 4. Property of a “renderer”

Displaying an animated image also plays the sounds it contains. Spatial transformation has audible effect; the horizontal component of translation becomes left/right balance, and the scale becomes volume adjustment. Cropping silences sounds outside of the crop region.

By definition, a *renderer* not only has the type above, but must also perform cropping, coloring, scaling, rotation, and time transformation, according to its given parameters, as expressed in Figure 4 (The “\$” operator is an alternative notation for function application. Because it is right-associative and has low syntactic precedence, it is sometimes used to eliminate cascading parentheses. Also, an infix operator (here ‘timeTransform’) with one argument but missing the other denotes a function that takes the missing argument (here an *ImageB* value) and fills it in.)

Fran defines convenience functions *move*, *stretch*, and *turn* that make 2D transform behaviors and apply them via “*%”.

In the early implementations of Fran, *ImageB* was simply defined to be *Behavior Image*, for a type *Image* of static images. This representation made for a very simple implementation, but it has a fundamental problem: the image structure of an *ImageB* value cannot be determined at the behavior level. It must first be sampled with some time *t* to extract a static image, whose structure can then be examined. To display an animation then, one must repeatedly sample it, and display the resulting static images. Consequently, the display computation cannot build and maintain data structures and system resources for parts of an animation. In particular, it cannot allocate one sprite for each *flipImage* and

```

data ImageB
= EmptyImage           -- transparent everywhere
| SolidImage           -- solid color
| FlipImage HFlipBook RealB -- page # behavior
| RenderImage Renderer -- text, 2D & 3D geometry
| SoundI      SoundB   -- embedded sound
| Over        ImageB   ImageB -- overlay
| TransformI Transform2B ImageB -- 2D transformed
| WithColorI ColorB ImageB -- colored
| CropI       RectB ImageB -- cropped
| CondI       BoolB ImageB ImageB -- conditional
| UntilI      ImageB (Event ImageB) -- reactivity
| TimeTransI ImageB TimeB -- time transformed

```

Fig. 5. Data type representing 2D image animations

`renderImage`, set them moving, and then update the motion paths incrementally. That is, the implementation cannot take advantage of a sprite engine.

It appears then that the modularity imposed by lifting requires an underlying presentation library to work in “immediate mode” (draw this now), rather than “retained mode” (build a model and repeatedly edit and redisplay it), to borrow terms from 3D graphics programming. In designing Fran, however, we targeted the upcoming generation of graphics accelerator cards, which we believe to be increasingly oriented toward retained mode.

For reasons given above, the `ImageB` is not represented in terms of a static `Image` type, but rather as a recursively defined data type, as shown in Figure 5. The functions in Figure 3 are defined as simple optimizations of the `ImageB` constructors from Figure 5. For instance, transforming the empty or solid image has no effect:

```

xf % EmptyImage = EmptyImage
xf % SolidImage = SolidImage
xf % im         = TransformI xf im

```

Although a programmer would not be likely to transform empty or solid images explicitly, such compositions arise at runtime due to modular programming style, as well as some of the reactivity optimizations discussed in 6. The constructors `CondI`, `UntilI`, and `TimeTransI` are used to define the overloaded functions `condB`, `untilB`, and `timeTransform` that apply to sound, 3D geometry, and all behaviors, as well as to image animations. The “`%`” operator is overloaded to apply to several types, as well.

3 A Temporal Sprite Engine

The primary purpose of the sprite engine is to scale, move, and overlay a sequence of images, and to do so at a very high and regular rate. Because animations may involve arbitrarily complex behaviors, and because garbage collection and lazy evaluation cause unpredictable delays, we could not meet this goal if the window refresh were implemented in Haskell or even invoked by a Haskell program. For these reasons, we designed and implemented a sprite engine in C++ that does no memory allocation and runs in its own thread.

3.1 Sprites and Sprite Trees

The sprite engine maintains an ordered collection of sprites, represented via a C++ class hierarchy. The classes representing individual sprites are as follows.

- **FlipSprite** has a bitmap that is selected from a “flip book” object. Each flip book contains a pointer to a bitmap stored in video memory, and information describing a rectangular array of images contained somewhere within the bitmap. This array is taken to be a linear sequence of consecutive “pages”. Flip books and the bitmap images to which they refer are immutable and may be shared among any number of flip sprites. The current page number, spatial transform, and cropping rectangle are all stored in the unshared sprites rather than the shared flip book.
- **RenderedSprite** has its own drawing surface and a method for replacing the surface with a new one. This sprite is used for all “rendered” images, such as text, 2D geometry, and camera-viewed 3D geometry.
- **SolidSprite** is a uniformly colored sprite, cropped but not spatially transformed.
- **SoundSprite** is an embedded sound, and is neither cropped nor transformed. It contains a pointer to (typically immutable) shareable sound data, plus its own unshared attributes for volume, frequency, and left/right balance.

A displayed animation could be represented by a list of individual sprites, in back-to-front order, so that upper (later) sprites are painted over lower (earlier) sprites. This representation is the simplest for display, but is awkward for editing. When an event occurs in an animation, an animated object may be introduced, removed, or replaced. Since the appearance of such an object may contain any number of sprites, the event may introduce, remove, or replace a whole contiguous subsequence of sprites. The sprite engine makes it easy and fast to edit the sprite list at a conceptual level by using a list of *sprite trees* rather than a list of individual sprites. The internal nodes of these trees correspond exactly to the points of structural mutability of the animation being displayed. (For Fran’s use, the points of mutability are generated from the reactivity construct “untilB”.) Correspondingly, there is a **SpriteTree** subclass **SpriteGroup** that contains a list of sprite trees and supports a method to replace the list, in its entirety, with another one.

For conditional animations, such as generated by Fran’s “`condB`”, there is a `SpriteTree` subclass `CondSpriteTree`, which contains an externally mutable boolean and two sprite tree lists.

The sprite engine expects its sprite trees to be updated less often than they are redisplayed. For example, Fran tries to sample behaviors at roughly ten times per second, but the sprite engine redispays at 60 times per second. Between updates, the sprite engine performs linear interpolation of all sprite attributes, which are actually represented as linear functions rather than constant values. For smoothness, updates must then be given for times in the future, so that a new linear path may be set from the current time and value to the given one. These linear behaviors form a compromise between arbitrary behaviors, which have unpredictable sampling requirements, and mere constant values, which fail to provide smooth motion. Other possible compromises include quadratic and cubic curves, for which fast incremental sampling algorithms may be used.

The main loop of the sprite engine is then to traverse its sprite tree list recursively and in depth-first, back-to-front order. Each image sprite in turn is told to paint itself, and in doing so samples its attributes’ linear functions and then does a very fast video memory “blit” (copy).

3.2 Interpretation of Sprite Trees

The goal of `ImageB` display is first to “spritify”, i.e., convert an abstract `ImageB` value to an initial list of sprite trees, and then update the trees iteratively. Concurrently, the sprite engine traverses and displays the sprite trees. Although the sprite trees are really implemented in C++, for the purpose of exposition, we will express them here in as the Haskell type definitions in Figure 6.

In order to define “correct” conversion, we need an interpretation of sprite tree lists. We specify this interpretation by mapping sprite tree lists to `ImageB` values. Note that this mapping is hypothetical, serving to (a) specify what the sprite engine does, and (b) justify our implementation in Section 4 of the *reverse* mapping, i.e., from `ImageB` values to `[SpriteTree]`. The interpretation functions are given in Figure 7. The function `treesIm` is the main one, saying that a list of sprite trees represents an overlay of the images represented by the member trees, in reverse order. The first three clauses of the `treeIm` function interpret individual sprites. Note the rigidity of order of applied operations imposed by the sprite constructors, as contrasted with the flexibility afforded by the `ImageB` type (Figure 3). It is exactly because of this difference that, while interpretation of sprite tree lists is straightforward, generation is not.

The `UntilT` case says that a reactive sprite tree represents a reactive image animation. This animation is initially the one represented by the initial sprite tree list. When the event occurs, yielding a new sprite tree list, the animation switches to the one represented by these new trees. (The event `e ==> treesIm` occurs whenever `e` does. At each occurrence, the function `treesIm` is applied to the the event data from `e`.) The conditional case is similar.

```

data SpriteTree =
    SoundSprite RealB          -- volume adjust
                    RealB      -- left/right pan
                    RealB      -- pitch adjust
                    Bool       -- whether to auto-repeat
                    SoundBuffer
  | RenderedSprite Vector2B    -- translation vector
                    SurfaceULB
  | SolidSprite RectB          -- cropping region
                    ColorB     -- solid color
  | FlipSprite RectB          -- cropping region
                    Vector2B    -- translation vector
                    RealB       -- scale factor
                    HFlipBook
                    RealB       -- page number
  | UntilT [SpriteTree] (Event [SpriteTree])
  | CondT  BoolB [SpriteTree] [SpriteTree]

```

Fig. 6. Sprite trees as Haskell types

4 From Abstract Animations to Sprite Trees

The interpretation of sprite tree lists as image animations given above helps to specify the reverse process, which we must implement in order to use the sprite engine to display animations: The `spritify` algorithm takes sprite trees to overlay, a cropping rectangle, optional color, a space- and a time-transformation, all applied to the given `ImageB` while spritifying. (Recall `mbWithColor` from Section 2)

```

spritify :: [SpriteTree] -> RectB -> Maybe ColorB -> Transform2B
          -> TimeB -> ImageB -> [SpriteTree]
treesIm (spritify above rectB mbColorB xfB tt imB) ==
  treesIm above 'over'
  (crop rectB          $
   mbWithColor mbColorB $
   (xfB *))           $
  ('timeTransform' tt) $
  imB)

```

The algorithm works by recursive traversal of `ImageB` values, accumulating operations that are found out of order. To get the algorithm started, Fran has just an `ImageB`, `imB`, and a cropping rectangle, `windowRectB`, based on the display window's size (which may vary with time), and so invokes `spritify` as `spritify [] windowRectB Nothing identity2 time imB`, where `identity2` is the identity 2D transform, and `time` serves as the identity time transform.

Our claim that the algorithm given below satisfies this specification may be proved by induction on the `ImageB` representation. Rather than give the

```

treesIm :: [SpriteTree] -> ImageB
treesIm []           = emptyImage
treesIm (bot : above) = treesIm above 'over' treeIm bot

treeIm (RenderedSprite motionB surfaceB) =
  move motionB (surfaceIm surfaceB)

treeIm (SolidSprite rectB colorB) =
  crop rectB (withColor colorB solidImage)

treeIm (FlipSprite rectB motionB scaleB book pageB) =
  crop rectB      $
  move motionB    $
  stretch scaleB $
  flipImage book pageB

treeIm (trees 'UntilT' e) = treesIm trees 'untilB' (e ==> treesIm)

treeIm (CondT c trees trees') = condB c (treesIm trees) (treesIm trees')

```

Fig. 7. Interpreting sprite trees

algorithm first and then the proof, however, we instead derive the algorithm from the specification so that it is correct by construction. We will rely on many simple algebraic properties of our data types.

Below we derive the `ImageB`-to-`[SpriteTree]` conversion for a small representative set of `ImageB` constructors. See [5] for a full treatment.

4.1 Solid Images

Solid images are unaffected by transformations, and are of a default color, as expressed by the following properties:

```

xfB *% SolidImage == SolidImage

SolidImage 'timeTransform' tt == SolidImage

withColor defaultColor SolidImage == SolidImage

```

These facts simplify the specification of `spritify`:

```

treesIm (spritify above rectB mbColorB xfB tt SolidImage) ==
  treesIm above 'over'
  crop rectB (withColor (chooseColorB mbColorB) SolidImage)

```

where


```

chooseColorB :: Maybe ColorB -> ColorB
chooseColorB Nothing  = defaultColor
chooseColorB (Just c) = c

```

Now considering the interpretation of `SolidSprite` and the definition of `treesIm` given in Figure 7, the specification simplifies further:

```

treesIm (spritify above rectB mbColorB xfB tt SolidImage) ==
  treesIm (SolidSprite rectB (chooseColorB mbColorB) : above)

```

This final simplification then directly justifies the `SolidImage` case in the `spritify` algorithm.

```

spritify above rectB mbColorB xfB tt SolidImage =
  SolidSprite rectB (chooseColorB mbColorB) : above

```

4.2 Rendered Images

Because of their spatially continuous nature, `ImageB` values may be thought of as being constructed bottom-up, whereas their implementation via `spritify` is top-down. For instance given “`stretch 10 circle`”, we would not want to render a circle to a (discrete) bitmap and then stretch the bitmap, because the resulting quality would be poor. Similarly, for “`stretch 0.1 circle`”, it would be wasteful to render and then shrink. Instead, scaling transforms are postponed and incorporated into the rendering, which can then be done at an appropriate resolution. Similarly, rotation is an expensive operation on bitmaps, but can generally be moved into rendering.

For this reason, we want to factor the transform behavior into translation, scale, and rotation. Fran’s `factorTransform2` function does just this:

```

xfB == translate motionB ‘compose2‘
      uscale    scaleB ‘compose2‘
      rotate    angleB
where
  (motionB, scaleB, angleB) = factorTransform2 xfB

```

Moreover, the meaning of transform composition is function composition.

Factoring the spatial transform and turning transform composition into function composition, our specification becomes the following.

```

treesIm (spritify above rectB mbColorB xfB tt
          (RenderImage renderer)) ==
  treesIm above ‘over‘
  (crop rectB          $
   mbWithColor colorB $
   move    motionB    $
   stretch scaleB     $
   turn    angleB     $
   (‘timeTransform‘ tt) $
   RenderImage renderer)
where
  (motionB, scaleB, angleB) = factorTransform2 xfB

```

Next, since renderers do not perform motion, we must extract the `move` from within applications of `withColor` and `crop`. Coloring commutes with spatial transformation:

```
withColor c (xfb % imb) == xfb % (withColor c imb)
```

Cropping is trickier, requiring that the cropping rectangle be inversely transformed:

```
crop rectB (xfb % imb) == xfb % crop (inverse2 xfb % rectB) imb
```

For example, doubling the size of an image and then cropping with a rectangle it is equivalent to cropping with a half-size rectangle and then doubling in size.

The definition of a *renderer* in Section 2 then leads to the following rule for spritifying rendered images. (The cropping and scaling behaviors should really be added to the `RenderImage` constructor and have the sprite engine use them, even though these operations are already performed during rendering. The reason is that the sprite engine can apply them incrementally at a much higher rate, for smoothness.)

```
spritify above rectB mbColorB xfb tt (RenderImage renderer) ==
  RenderedSprite motionB
    (renderer (move (-motionB) rectB)
      mbColorB scaleB angleB tt)
: above
where
  (motionB, scaleB, angleB) = factorTransform2 xfb
```

4.3 Overlays

The treatment of overlays follows from the distribution of time transformation, space transformation, coloring, and cropping over the `over` operation, plus the associativity of `over`.

```
spritify above rectB mbColor xfb tt (top 'Over' bot) =
  spritify (spritify above rectB mbColor xfb tt top)
    rectB mbColor xfb tt bot
```

4.4 Space Transformation

Time transformation distributes over space transformation:

```
(xfb % imb) 'timeTransform' tt ==
(xfb 'timeTransform' tt) % (imb 'timeTransform' tt)
```

Space transforms compose:

```
xfb % (xfb' % imB) == (xfb 'compose2' xfb') % imB
```

The rule for `TransformI` then follows easily.

```
spritify above rectB mbColor xfb tt (xfb' 'TransformI' imb) =
  spritify above rectB mbColor
    (xfb 'compose2' (xfb' 'timeTransform' tt)) tt imb
```

4.5 Conditional Animation

Time- and space-transformation, coloring, and cropping, all distribute over conditionals:

```

(condB boolB thenB elseB) 'timeTransform' tt
== condB (boolB 'timeTransform' tt)
      (thenB 'timeTransform' tt)
      (elseB 'timeTransform' tt)

xf *% (condB boolB thenB elseB)
== condB boolB (xf *% thenB) (xf *% elseB)

withColor c (condB boolB thenB elseB)
== condB boolB (withColor c thenB) (withColor c elseB)

crop rectB (condB boolB thenB elseB)
== condB boolB (crop rectB thenB) (crop rectB elseB)

```

The rule then follows:

```

spritify above rectB mbColor xfB tt (CondI c imb imb') =
  CondT (c 'timeTransform' tt)
        (spritify [] rectB mbColor xfB tt imb)
        (spritify [] rectB mbColor xfB tt imb)
  : above

```

4.6 State and Concurrent Updating

The spritifying algorithm above is idealized in that it constructs immutable sprite tree lists containing Fran behaviors. In fact, the sprite engine's data structures are mutable, both parametrically (e.g., sprite position and size) and structurally (e.g., number and kind of sprite), and only accomodate linear behaviors. The actual implementation of `spritify` creates the mutable sprite trees with initial values for position, scale, pitch, etc., and then iteratively updates these attributes, while the sprite engine runs concurrently. For simplicity of implementation, every active sprite is managed by its own Haskell thread, using the Concurrent Haskell primitives [10]. Each such thread is fueled by a request channel, with each request saying either to continue or to quit, and puts status messages into a response channel. Each sprite thread iteratively samples the appropriate behaviors (slightly into the future) and invokes an update method on the sprite object. In response, the sprite engine charts a new linear course for each attribute.

For example, Figure 8 is a slightly simplified implementation of the `SolidImage` case. A monochrome sprite is allocated in the sprite engine, chaining to the sprite trees given as “above”. The time stream `ts` is used to sample the cropping rectangle and color, using `ats` (which is memoized as described in [6]). The resulting sample values are used to interactively update the sprite attributes, as long as the request channel `requestV` says to continue. The iteration is placed in a new Haskell thread via `forkIO`.

```

spritify :: SpriteTree -> RectB -> Maybe ColorB
        -> Transform2B -> ImageB -> Maybe TimeB
        -> [Time] -> MVar Bool -> MVar Bool -> IO SpriteTree

spritify above rectB mbColorB xfB tt SolidImage
    ts requestV replyV = do
    sprite <- newMonochromeSprite above
    let update ~(t:ts')
        ~(RectLLUR (Point2XY llx lly) (Point2XY urx ury) : rects')
        ~(ColorRGB r g b : colors') = do
    continue <- takeMVar requestV
    if continue then do
        updateMonochromeSprite sprite t llx lly urx ury r g b
        putMVar replyV True
        update ts' rects' colors'
    else
        putMVar replyV False
    forkIO $ update ts (rectB 'ats' ts) (chooseColorB mbColorB 'ats' ts)
    return (toSpriteTree sprite)

```

Fig. 8. `spritify` case with concurrent updating

The handling of a reactive image animation, `imb 'untilB' e`, is somewhat tricky. The initial animation `imb` is spritified, starting any number of threads. The resulting sprite tree list is wrapped up in a `SpriteGroup` object (corresponding to the `UntilT` constructor above). One more thread runs to watch for event occurrences and causes the update work corresponding to `imb` to continue until the first occurrence of the event `e`, and then to stop. At that point, a new animation is available and is spritified, generating a new sprite tree list, which is then passed to a method on the `SpriteGroup` object that recursively deletes its old list and installs the new one. The number of running threads thus varies in response to events. Because every thread has accompanying overhead for controlling its work, a useful future optimization would be to create many fewer threads.

5 Conclusions

The implementation techniques described in this paper bridge the gap between functional animation and retained-mode display. Functional animation serves the needs of composability, while retained-mode promotes efficient use of hardware resources. A recurring theme in this work is the application of algebraic properties of our animation data types, in order to normalize animations to the relatively restrictive form imposed by retained-mode presentation libraries. An area of future work is to develop rigorous semantic models for these data types.

The models would form the interface between the informal mental models taught to and used by the everyday animation programmer and the correct and efficient implementation of the animation library itself. This semantic orientation has driven our work from the start, but some work remains to make it complete and precise.

References

- [1] Kavi Arya. A functional animation starter-kit. *Journal of Functional Programming*, 4(1):1–18, January 1994.
- [2] Conal Elliott. A brief introduction to ActiveVRML. Technical Report MSR-TR-96-05, Microsoft Research, 1996. <http://research.microsoft.com/scripts/pubDB/pubsasp.asp?RecordID=38>.
- [3] Conal Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *The Conference on Domain-Specific Languages*, pages 285–296, Santa Barbara, California, October 1997. USENIX. WWW version at <http://research.microsoft.com/~conal/papers/ds197/ds197.html>.
- [4] Conal Elliott. Composing reactive animations. *Dr. Dobb's Journal*, July 1998. Extended version with animations at <http://research.microsoft.com/~conal/fran/{tutorial.htm,tutorialArticle.zip}>.
- [5] Conal Elliott. From functional animation to sprite-based display (extended version). Technical Report MSR-TR-98-28, Microsoft Research, 1998. <http://research.microsoft.com/scripts/pubDB/pubsasp.asp?RecordID=190>.
- [6] Conal Elliott. Functional implementations of continuous modeled animation. In *Proceedings of PLILP/ALP '98*, 1998. <http://research.microsoft.com/~conal/papers/plilpalp98/short.ps>. Extended version, MSR-TR-98-25, <http://research.microsoft.com/scripts/pubDB/pubsasp.asp?RecordID=164>.
- [7] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 263–273, Amsterdam, The Netherlands, 9–11 June 1997. <http://research.microsoft.com/~conal/papers/icfp97.ps>.
- [8] Paul Hudak and Joseph Fasel. A gentle introduction to Haskell. *SIGPLAN Notices*, 27(5), May 1992. See <http://haskell.org/tutorial/index.html> for latest version.
- [9] Paul Hudak, Simon L. Peyton Jones, and (editors) Philip Wadler. Report on the programming language Haskell, A non-strict purely functional language (Version 1.2). *SIGPLAN Notices*, Mar, 1992. See <http://haskell.org/report/index.html> for latest version.
- [10] Simon Peyton Jones, Andrew Gordon, and Sigbjørn Finne. Concurrent Haskell. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 21–24 January 1996.
- [11] John Peterson, Conal Elliott, and Gary Shu Ling. Fran user's manual, Revised February, 1998. <http://research.microsoft.com/~conal/Fran/UsersMan.htm>.

Beyond Pretty-Printing: Galley Concepts in Document Formatting Combinators

Wolfram Kahl

Institut für Softwaretechnologie, Fakultät für Informatik
Universität der Bundeswehr München, D-85577 Neubiberg, Germany
E-Mail: kahl@informatik.unibw-muenchen.de

Abstract. Galleys have been introduced by Jeff Kingston as one of the key concepts underlying his advanced document formatting system Lout. Although Lout is built on a lazy functional programming language, galley concepts are implemented as part of that language and defined only informally.

In this paper we present a first formalisation of document formatting combinators using galley concepts in the purely functional programming language Haskell.

1 Introduction and Related Work

Pretty printing is an established topic in the functional programming community; it is targeted towards appropriate layout of data structures, with one of the main applications being user-friendly display of internal data structures of compilers, for example for the creation of error messages. Hughes has turned the development of pretty-printing combinators into a fine art in [2,3]. His combinators have subsequently been improved upon by Peyton Jones [10]. Wadler [11] has presented a simpler design; yet another set of “optimal pretty printers” has come forth in [1].

However, document formatting with problems such as the placement of footnotes is simply outside the scope of pretty printing as discussed in the papers mentioned so far.

The best-known document formatting system in the scientific community is probably T_EX [9]. T_EX is built on a rather large number of primitives that incorporate many fine points of the art of typesetting. T_EX offers access to these primitives through an imperative programming language with dynamic binding and rather weak structuring capabilities — in short, T_EX considered as a programming language is about as impure as possible.

Another document formatting system recently gaining acceptance and popularity is Jeff Kingston’s Lout [6,7,8]. Lout had twenty-three primitives in the version presented in [6] — this number has grown in the meantime, but is still small. The programming language available for programming document format definitions is a small lazy functional programming language with an innovative definition and name visibility mechanism, flexible composition of aligned objects, a versatile cross reference concept and, most important for this paper, the *galley* concept.

The galley abstraction allows to direct part of the document to places somewhere else in the document, for example footnotes are implemented as galleys directed to targets at the bottom of the page. The generality of the galley abstraction as conceived by

Kingston allows essentially all problems of text flow, tables of contents, and even indices and lists of references to be defined via galleys, so that Lout is a document formatting system that does not even need a built-in concept of pages — pages are defined in the programming language as objects of a certain size and containing certain targets.

However, the galley concept itself is defined outside the functional programming language of Lout, described only informally in [6] (and even more informally in the “Expert’s Guide” [7]), and implemented in C.

In this paper we give a completely formal definition of galley movement in the pure functional programming language Haskell. By disregarding all aspects unrelated to galley flushing we are able to present a small set of basic document combinators that satisfactorily capture the essential behaviour of galleys in Lout and still can be implemented with relative ease.

2 Overview of Document Formatting Combinators

We start with giving a brief overview of how to reflect the central features of Lout’s galleys in document formatting combinators.

The exposition of this paper is centred around *objects*, the datatype for which we shall present in detail in the next section:

```
> data Object
```

A very important object is the `null` object which never leaves a trace¹:

```
> null :: Object
```

For being able to build up more interesting documents, we have to decide on their basic constituents. Adhering to Lout nomenclature we call these *components*. For this paper, we consider lists of lines as components — this is more general than just lines and allows us to experiment with unbreakable components of different heights:

```
> type Component = [String]
```

We want to be able to turn components into objects, and to prefix objects with components:

```
> singleton :: Component -> Object
```

```
> prefix    :: Component -> Object -> Object
```

Obviously we can define `singleton` via `prefix`:

```
> singleton c = prefix c null
```

A more general means to compose objects is *concatenation*:

```
> (#) :: Object -> Object -> Object
```

In the expert’s guide to Lout [7], Kingston defines: “A *galley* is an object plus a cross reference which points to where the object is to appear.” However, cross references occurring in galleys are limited to those referring to a name together with a direction²:

```
> type Galley = (String, Direction, Object)
```

```
> data Direction = Preceding | Following
```

Such a cross reference is taken to mean the list of all targets of the same name, starting with the next occurrence in the respective direction and always continuing forward.

¹ Since we have a name clash here, we have to remember to “> import Prelude hiding (null)”!

² In Lout, a third mode is possible which is only used for bibliographic reference list entries.

A galley considered as an object sends itself as the first outgoing galley and otherwise behaves like `null` — in Lout, such a galley is built by defining a symbol with the galley object as the body of the definition and a special “@Target” parameter (or the “into” abbreviations).

```
> galley :: Galley -> Object
```

In Lout, a target that is receptive to galleys is built using the “@Galley” symbol, which therefore does not represent a galley, but a possible *target* for galleys. Our combinator for introducing targets is the following:

```
> target :: String -> Object
```

These combinators are not yet sufficient for the definition of pages. Obviously a page should contain targets for text and footnotes, but the essential feature of a page is that it restricts the size of what can be received into these targets. Size restriction is achieved by using the following combinator:

```
> high :: Int -> Object -> Object
```

The footnote section, when it appears, will start with some space and a horizontal line followed by a list of footnotes — we can achieve this by “prefix [“”, “----”] footList”.

However, when there is no footnote, then the whole footnote section should disappear — Lout has the concept of *receptive symbols* for this: A receptive symbol is any symbol defined to directly or indirectly contain a target, and receptive symbols are replaced by `null` if no contained target ever receives a galley.

Our framework lacks the concepts of symbols and definitions, therefore we make the effect of receptive symbols explicit as a combinator that lets its argument object appear only if it receives a galley:

```
> delay :: Object -> Object
```

Lout relies on recursive definitions for implementing e.g. lists of pages or footnotes; however Lout uses a special semantics of recursion that includes recursive objects being delayed and not expanding unless space is sufficient. Since we cannot change the semantics of recursion in Haskell, we have to provide our own combinator for building recursive objects and abstain from using Haskell recursion in object definitions:

```
> recurse :: (Object -> Object) -> Object
```

Finally we need a function that actually formats a document; its result is a list of components, which usually shall stand for pages:

```
> force :: Object -> [Component]
```

3 A Simple Document Format

These combinators are sufficient to build document formats analogous to those used to explain galley concepts in [6,7].

A list of pages to hold the whole document is a simple recursion over of a page concatenated with the recursive call, and every page is a restricted-height object containing a target for text concatenated with a footnote section:

```
> pageList = recurse (page #)
```

```
> page = high 12 (target "TextPlace" # footSect)
```


The footnote section is delayed, and contains a recursive list of footnote targets (in Haskell, the infix operator “\$” is low-priority, right-associative function application):

```
> footSect = delay $ prefix ["", "-----"] footList
> footList = recurse (target "FootPlace" #)
```

Now we have two kinds of targets; in the simple documents we are considering, correspondingly two kinds of galleys occur, the main text of the document and footnotes:

```
> text    t = galley ("TextPlace", Preceding, t)
> footNote f = galley ("FootPlace", Following, f)
```

A sample document is constructed with ease:

```
> purcell = prefix ["PURCELL(1)"] (footNote fn) # body
> fn = singleton ["(1) Blom, Eric. Some", "Great Composers.", "Oxford, 1944."]
> body = foldr prefix null [ ... ]
```

So now we can format the running example of [6,7] by evaluating the expression: `force (pageList # text purcell)`, and displaying the result in a pleasing manner:

PURCELL(1)	that is regarded	of the world's musical
In the world of music	elsewhere as perfectly	classics, as here, we find
England is supposed to be	normal and natural; but	that we cannot omit this
a mere province. If she	if foreign students of	English master.
produces an indifferent	musical history have to	
composer or performer,	acknowledge a British	
-----	musical genius, he is	
(1) Blom, Eric. Some	considered a freak.	
Great Composers.	Such a freak is	
Oxford, 1944.	Henry Purcell. Yet if we	
	make a choice of fifteen	

4 Implementation

Objects are modeled as entities that essentially are prepared to react to two kinds of stimuli:

- an *evaluation request* tries to extract a component into available space,
- a *reception request* asks the object to receive a galley directed at some target contained in the object — we consider it to be an error to send a galley to an object that does not contain a target for it.

The reactions to both of these have to take into account the available space, which is expressed as a *Constraint*, and a boolean *forcing* parameter which influences the eagerness of the object's behaviour and depends on the object's position inside concatenation chains.

Furthermore we need to know which receiving targets an object contains, both open and within delayed or recursive subobjects, and its minimum height as a whole. Therefore we define:

```
> data Object = Object
> {eval      :: Bool -> Constraint -> EvalResult
> ,receive   :: Bool -> Constraint -> Galley -> RcvResult
> ,openTargets :: [String]
> ,delayedTargets :: [String]
> ,height    :: Int
> }
```

```
> type RcvResult = (Bool, [Galley], Object)
> targets o = openTargets o ++ delayedTargets o
```

The result of evaluating can be:

- Disappearing: The object disappears without trace, which is what `null` always does.
- Suspended: The object cannot yet say what it is going to be, as for example targets that have not yet been attached to, or recursive or delayed objects. If `forcing` is set to `True` in the call to `eval`, an object must not suspend — it will then usually choose to disappear.
- NoSpace: The object wants to yield a first component which is too large for the current constraints.
- Sending: A list of galleys is dispatched.
- Yielding: A non-empty component fitting into the current space constraints is yielded.

Except Disappearing, all results carry a continuation object since the evaluation attempt always might change the object:

```
> data EvalResult = Disappearing
>                  | Suspended { obj :: Object }
>                  | NoSpace   { obj :: Object }
>                  | Sending   { gall :: [Galley], obj :: Object }
>                  | Yielding  { comp :: Component, obj :: Object }
```

The only measure to be constrained in our current setting is the height of objects; and there is also the possibility that an object is evaluated without any constraints, so we define:

```
> type Constraint = Maybe Int
```

For modelling more of Lout, the constraint datatype should not only contain information about the allowed height and width, but also other settings such as font and spacing information.

With this we can already define the global document formatting function which forces evaluation and does not impose any constraints — we decide to simply discard any stray galleys:

```
> force :: Object -> [Component]
> force o = case eval o True Nothing of
>   Disappearing -> []
>   Yielding c o' -> c : force o'
>   Sending gs o' -> force o'
```

The possible results of `receive` have the following semantics:

- “`(True,gs,o)`” means that the galley was received, triggered further galleys `gs`, and that the resulting object is “`o`”.
- “`(False,gs,o)`” means that the galley was *not* received, usually because of insufficient space, but that the object has triggered the galleys `gs` and may have undergone changes (such as deciding never to receive anymore); therefore one now has to refer to “`o`” instead. Although the argument galley has not been received, it may have been modified or discarded, so if it is to be sent on, it will always be found in `gs`.

The `null` object is always ready to disappear immediately without leaving any trace (`rcverror` documents an error that should not occur since objects should only be sent galleys they can receive):

```

> null :: Object
> null = Object
> {eval = (\ forcing co -> Disappearing)
> ,receive = (\ forcing co g -> rcverror g "null")
> ,openTargets = []
> ,delayedTargets = []
> ,height = 0
> }

```

A singleton object yields its component if there is space for it and otherwise signals NoSpace. For testing whether some component or object fits into a constraint, respectively for decreasing a constraint by the amount indicated by some component or object, we use an appropriately defined class Constrainer and the following functions:

```

> (&?) :: Constrainer c => Constraint -> c -> Bool
> (&-) :: Constrainer c => Constraint -> c -> Constraint

```

We give an explicit definition of singleton:

```

> singleton :: Component -> Object
> singleton c = o where
>   o = Object
>   {eval = (\ forcing co -> if co &? c then Yielding c null else NoSpace o)
>   ,receive = (\ forcing co g -> rcverror g "singleton")
>   ,openTargets = [], delayedTargets = []
>   ,height = length c
>   }

```

The same space considerations apply to prefix, which otherwise inherits the behaviour of the prefixed object:

```

> prefix :: Component -> Object -> Object
> prefix c o = o' where
>   o' = Object
>   {eval = (\ forcing co -> if co &? c then Yielding c o else NoSpace o')
>   ,receive = (\ forcing co g ->
>               thrdUpd3 (prefix c) $ receive o forcing (co &- c) g)
>   ,openTargets = openTargets o
>   ,delayedTargets = delayedTargets o
>   ,height = length c + height o
>   }

```

For updating the object within the RcvResult we used the prelude-quality thrdUpd3:

```

> thrdUpd3 :: (c -> d) -> (a, b, c) -> (a, b, d)
> thrdUpd3 f (a, b, c) = (a, b, f c)

```

It is easy to see that the equation `singleton c = prefix c null` does indeed hold.

A galley considered as an object sends itself as the first outgoing galley and otherwise behaves like null:

```

> galley :: Galley -> Object
> galley g = Object
>   {eval = (\ forcing co -> Sending [g] null)
>   ,receive = (\ forcing co g' -> rcverror g' "galley")
>   ,openTargets = [], delayedTargets = []
>   ,height = 0
>   }

```

Before we start to present the definition of object concatenation, we first introduce an auxiliary combinator that shall be needed there. Taking a component `c` and an object `o`, suffix delivers an object that yields `c` only after `o` disappears:

```

> suffix :: Component -> Object -> Object
> suffix c o = o' where
>   o' = Object
>   {eval = (\ forcing co -> case eval o forcing (co &- c) of
>       Disappearing -> eval (singleton c) forcing co
>       r -> r {obj = suffix c (obj r)})}
>   ,receive = (\ forcing co g ->
>       thrdUpd3 (suffix c) $ receive o forcing (co &- c) g)
>   ,openTargets = openTargets o
>   ,delayedTargets = delayedTargets o
>   ,height = length c + height o
>   }

```

Concatenation of two objects into a single object has two main tasks:

- *Communication*: Galleys sent off by one object, according to their direction either have to be sent to the other object or to the context.
- *Space negotiation*: Two objects placed together within one constraint may only grow as long as they do not infringe into each others space.

Since the final result of evaluating an object must always either be that the object disappears or that it yields a component *from the object's beginning*, a natural asymmetry is imposed on both seemingly symmetrical tasks. We resolve this asymmetry using the forcing parameters to eval and receive as indicated by the following:

- While galleys are sent between the several objects in a concatenation tree, galleys arriving at the left-most leaf are not expanded, while all other galleys are expanded as far as possible (are “forced”) on arrival. This ensures for example that footnotes are allowed to grow after being sent off before the subsequent main text (which is sent to the competing “TextPlace” target) is allowed to grow into its space.
- While objects are evaluated, Suspended objects are left waiting everywhere except in the right-most leaf. There, evaluation is “forced”, and targets or delayed objects are made to disappear — if no galley has reached them yet, there will also be no such galley later on. Otherwise we would never be able to determine the end of a document in our sample format, since there is always a `pageList` waiting at the right end.

For communication of galleys we first present a few auxiliary definitions. We use a communication state that contains two objects and a flag signalling whether some galley already has been received:

```

> type OCState = (Bool, Object, Object)

```

This is then used as the state for a simple state monad:

```

> type STfun s x = s -> (s,x)
> type SToc = STfun OCState [Galley]

```

We provide a function to combine results of `SToc` transformers into `RcvResults`:

```

> ocMkResult :: (OCState,[Galley]) -> RcvResult
> ocMkResult ((rcv, o1, o2), gs) = (rcv, gs, o1 # o2)

```

Communication between two adjacent objects has to respect the direction of the galleys sent off by either. Furthermore we take care that only the very first object of a longer concatenation receives in a delayed manner, while all later objects receive with forcing set. The definitions of `ocGalleyFrom1` and `ocGalleyFrom2` are therefore symmetric except for the forcing parameter passed to the receive calls: The second object of a concatenation always receives with `forcing = True`, while the first inherits from above:

```

> ocGalleyFrom1, ocGalleyFrom2 :: Bool -> Constraint -> Galley -> SToc
> ocGalleyFrom1 forcing co g @ (name,Preceding,_) s = (s, [g])
> ocGalleyFrom1 forcing co g @ (name,Following,_) s @ (rcv, o1, o2) =
>   if name 'notElem' targets o2 then (s,[g])
>   else let (rcv', gs2, o2') = receive o2 True (co &- o1) g
>         in ocGalleysFrom2 forcing co gs2 (rcv || rcv', o1, o2')
> ocGalleyFrom2 forcing co g @ (name,Following,_) s = (s, [g])
> ocGalleyFrom2 forcing co g @ (name,Preceding,_) s @ (rcv, o1, o2) =
>   if name 'notElem' targets o1 then (s,[g])
>   else let (rcv', gs1, o1') = receive o1 forcing (co &- o2) g
>         in ocGalleysFrom1 forcing co gs1 (rcv || rcv', o1', o2)

```

Iterated communication is achieved via a simple fold function in the state monad:

```

> stfold :: (a -> STfun b [c]) -> [a] -> STfun b [c]
> stfold f []      s = (s, [])
> stfold f (g:gs) s = let (s' , gs' ) = f g s
>                        (s'', gs'') = stfold f gs s'
>                        in (s'', gs' ++ gs'')
> ocGalleysFrom1, ocGalleysFrom2 :: Bool -> Constraint -> [Galley] -> SToc
> ocGalleysFrom1 forcing co = stfold (ocGalleyFrom1 forcing co)
> ocGalleysFrom2 forcing co = stfold (ocGalleyFrom2 forcing co)

```

In the definition of `receive`, the concatenation looks at the open and delayed receptors of both components and decides accordingly to which the galley is going to be sent.

```

> (#) :: Object -> Object -> Object
> o1 # o2 = o where
>   o = Object
>   {eval = ocEval o1 o2
>   ,receive = (\ forcing co g @ (name,d,o') -> let
>       send1 = let (r,gs,o1') = receive o1 forcing (co &- o2) g
>               in ocMkResult $ ocGalleysFrom1 forcing co gs (r, o1', o2)
>       send2 = let (r,gs,o2') = receive o2 True (co &- o1) g
>               in ocMkResult $ ocGalleysFrom2 forcing co gs (r, o1, o2')
>       sendO1 x = if name 'elem' openTargets o1 then send1 else x
>       sendO2 x = if name 'elem' openTargets o2 then send2 else x
>       sendD1 x = if name 'elem' delayedTargets o1 then send1 else x
>       sendD2 x = if name 'elem' delayedTargets o2 then send2 else x
>       in case d of
>         Following -> sendO1 $ sendO2 $ sendD1 $ sendD2 $ rcverror g "(#)"
>         Preceding -> sendO2 $ sendO1 $ sendD2 $ sendD1 $ rcverror g "(#)"
>   ,openTargets = openTargets o1 ++ openTargets o2
>   ,delayedTargets = delayedTargets o1 ++ delayedTargets o2
>   ,height = height o1 + height o2
>   }

```

When evaluating the concatenation of two objects, we first evaluate the first (never forcing) and then perform any communication that might result from the galleys sent by that evaluation. Only the case that the first object suspends is not straight-forward. In that case, we always evaluate the second object, too, in the hope that the resulting communication relieves the suspension. Therefore we resume evaluation of the combined object if the concatenated object is forced or if either communication succeeded or if the evaluation of the second object yielded a component — in order to enable more of the second object to be evaluated if necessary we have to use `suffix` to stick that component onto the end of the first object in the recombination.

```

> ocEval :: Object -> Object -> Bool -> Constraint -> EvalResult
> ocEval o1 o2 forcing co = case eval o1 False (co &- o2) of
>   Disappearing -> eval o2 forcing co
>   NoSpace o1' -> NoSpace (o1' # o2)
>   Yielding c o1' -> Yielding c (o1' # o2)
>   Sending gs o1' ->
>     case ocMkResult $ ocGalleysFrom1 False co gs (False, o1', o2) of
>       (rcv, [], o') -> eval o' forcing co
>       (rcv, gs, o') -> Sending gs o'
>   Suspended o1' -> case eval o2 forcing (co &- o1') of
>     Disappearing -> if forcing then eval o1' forcing co else Suspended o1'
>     Suspended o2' -> Suspended (o1' # o2')
>     NoSpace o2' -> if forcing then NoSpace o2' else Suspended (o1' # o2')
>     Yielding c o2' -> eval (suffix c o1' # o2') forcing co
>     Sending gs o2' ->
>       case ocMkResult $ ocGalleysFrom2 False co gs (False, o1', o2') of
>         (True, [], o') -> eval o' forcing co
>         (False, [], o') -> error ("empty Sending??")
>         (_, gs, o') -> Sending gs o'

```

The function `high` constrains the height of its object argument to the amount indicated by the integer argument. If the object argument does not fit in the indicated height, the remainder object is discarded and a placeholder of the appropriate height is substituted. Otherwise, the result is filled to the specified height.

Filling up to the specified height involves recursively evaluating the remaining objects after `Yielding` and concatenating the components into a single component — this task is delegated to the auxiliary function `prefixConc`. Besides we also use the following small functions:

```

> strut h = replicate h ""
> fill h c = take h (c ++ repeat "")

```

If the resulting object itself is placed in a too tightly constrained environment, then it does not fit and remains as the continuation object of `NoSpace`.

```

> high :: Int -> Object -> Object
> high h o = o' where
>   eval' forcing = case eval o forcing (Just h) of
>     NoSpace o1 -> Yielding (fill h ["@High: Object too large"]) null
>     Disappearing -> Yielding (strut h) null
>     Suspended o1 -> Suspended (high h o1)
>     Sending gs o1 -> Sending gs (high h o1)
>     Yielding c o1 -> let h' = h - length c in
>       if h' < 0 then error "@High: yielded component too high!"
>       else case eval (high h' o1) forcing Nothing of
>         Yielding c' o2 -> Yielding (c ++ c') o2
>         Sending gs o2 -> Sending gs (prefixConc c o2)
>         Suspended o2 -> Suspended (prefixConc c o2)
>         NoSpace o2 -> error "@High: NoSpace in recursive call!"
>         Disappearing -> Yielding (fill h c) null
>   o' = Object
>   {eval = (\ forcing co -> case co of
>     Nothing -> eval' forcing
>     Just h' -> if h' < h then NoSpace o' else eval' forcing)
>   ,receive = (\ forcing co g ->
>     thrdUpd3 (high h) $ receive o forcing (Just h) g)

```

```

> ,openTargets = openTargets o
> ,delayedTargets = delayedTargets o
> ,height = h
> }

```

The auxiliary function `prefixConc` used above is the tool that allows `high` to assemble components of exactly the right height from objects that yield small irregular components by modifying its argument object to *concatenate* the argument component before its first yielded component:

```

> prefixConc :: Component -> Object -> Object
> prefixConc c o = o' where
>   o' = Object
>   {eval = (\ forcing co -> case eval o forcing (co &- c) of
>     Disappearing -> Yielding c null
>     Yielding c' o2 -> Yielding (c ++ c') o2
>     r -> r {obj = prefixConc c (obj r)})}
> ,receive = (\ forcing co g -> if co &? c
>   then thrdUpd3 (prefixConc c) $ receive o forcing (co &- c) g
>   else (False, [forward g], o'))
> ,openTargets = openTargets o, delayedTargets = delayedTargets o
> ,height = length c + height o
> }

```

If a galley cannot be received because of lack of space, it has to be sent on looking for its next target; but it always has to be turned forward for this purpose (imagine overflowing text in the example presented in Sect. 3; although it finds its first page initially as the Preceding target, the next target to be unveiled by `pagelist` is the Following):

```

> forward :: Galley -> Galley
> forward (name,d,o) = (name,Following,o)

```

A galley that has reached its target transforms this target into a concatenation consisting of the object carried by the galley and the target itself which has to be ready to receive more galleys directed at it. However, if the galley object does not fit into the space at the target — a fact that cannot be excluded at reception time — it has to be sent on to the next target with the same name, and the original target disappears.

For this purpose we provide the following auxiliary combinator that shall be used in the definition of `target` and that *attaches* a received object to a target name.

An important case to consider is that the attached object might be a concatenation, since concatenation takes into account the heights of *both* its components before deciding to yield a component. However, if the attached object is too large for the space where it currently resides, it will send itself to the next target — therefore we have to evaluate the attached object with no size constraint so that it may yield its first component without internal size considerations (`isEmpty` determines whether a constraint is `Just 0`):

```

> attach :: String -> Object -> Object
> attach name = attach' where
>   attach' o = o' where
>     o' = Object
>     {eval = (\ forcing co -> case eval o forcing Nothing of
>       Disappearing -> Disappearing
>       NoSpace o1 -> error "attach: NoSpace without constraints!"
>       Suspended o1 -> if isEmpty co
>         then Sending [(name,Following,attach' o1)] null
>         else Suspended (attach' o1)}

```

```

>     Sending gs o1 -> Sending gs (attach' o1)
>     Yielding c o1 -> if co &? c then Yielding c (attach' o1)
>         else Sending [(name,Following,attach' (prefix c o1))] null)
> ,receive = (\ forcing co g -> thrdUpd3 attach' $ receive o forcing co g)
> ,openTargets = openTargets o
> ,delayedTargets = delayedTargets o
> ,height = 0
> }

```

The target function is used to build the basic receptive objects. It therefore disappears when evaluation is forced on it, and suspends otherwise.

When receiving, we have to distinguish whether reception is forced or not — remember that all but the first object of a concatenation have to perform forced reception. The motivation behind this is that, for example, the size of a footnote needs to be known before the text can be allowed to fill the remainder of the page. Therefore the footnote has to expand as far as possible once its galley has reached its target. Again, this expansion involves evaluation without constraints since the part that does not fit into the current target will be sent on to the next target.

```

> target :: String -> Object
> target name = o where
>   o = Object
>   {eval = (\ forcing co -> if forcing then Disappearing
>     else case co of Just 0 -> Disappearing
>     _ -> Suspended o)
>   ,receive = (\ forcing co g @ (name',d',o') -> case co of
>     _ -> if name /= name' then rcverror g "target"
>     else if not forcing then (True, [], (attach name o' # o))
>     else case eval o' False Nothing of
>       Disappearing -> (True, [], o)
>       Suspended o'' -> (True, [], (attach name o'' # o))
>       NoSpace o'' -> error "target: NoSpace without constraint!"
>       Sending gs1 o'' -> (True, gs1, (attach name o'' # o))
>       Yielding c o'' -> if co &? c then
>         let g' = (name',Following,o'')
>         (rcv, gs1, o1) = receive (prefix c o) forcing co g'
>         in (True, gs1, o1)
>     else (False, [(name,Following,prefix c o'')], null))
>   ,openTargets = [name]
>   ,delayedTargets = []
>   ,height = 0
> }

```

The last line of the definition of `receive` reflects the fact that when there is not enough space for the first component of the galley arriving at its target, then the galley is sent on to the next target in forward direction, and the original target disappears.

Objects built with `recurse` and `delay` are both delayed objects¹ and expand only if some contained target receives a galley. Therefore both disappear when forced and suspend on normal evaluation.

As a small concession to the forcing galleys of Lout (which are not directly

¹ In this paper we only consider *receptive* recursive objects. The behaviour of other recursive objects in Lout apparently cannot be integrated easily into the forcing strategy of the current paper.

connected with our forcing parameters) we allow them to disappear, too, when it is clear that there is not enough space for expansion. Since eventually these objects would be forced away in the end, this is not strictly necessary, but in the effect it allows pages to be flushed earlier.

When a recursive object is asked to receive a galley, it first checks whether a single instance of the body of the recursion, with `null` substituted for the recursive call, can receive the galley under the present size constraints. Only if this is possible, the recursion is unfolded one level and the galley sent to the result — note that this tentative evaluation is possible without problems since it cannot provoke any side effects. If reception is impossible, then the galley is sent on and the recursive object disappears — otherwise footnotes might appear out of order.

```
> recurse :: (Object -> Object) -> Object
> recurse ff = o
> where
>   ffo = ff o
>   ff0 = ff null
>   targs = targets ff0
>   o = Object
>     {eval = (\ forcing co -> if forcing || isEmpty co || not (co &? ffo)
>                                     then Disappearing else Suspended o)
>   ,receive = (\ forcing co g @ (name,d,o') -> case co of
>       Just 0 -> (False, [forward g], null)
>       _ -> if name 'elem' targs
>           then case receive ff0 forcing co g of
>               (False, gs, ol) -> (False, [forward g], null)
>               r -> receive ffo forcing co g
>           else rcverror g "recurse")
>   ,openTargets = []
>   ,delayedTargets = targs
>   ,height = 0
>   }
```

Since it only receives after expansion, a recursive object contains no open targets, and all targets of the body are delayed targets of the recursion.

For objects built with `delay`, the same considerations apply:

```
> delay :: Object -> Object
> delay o = o' where
>   o' = Object
>     {eval = (\ forcing co -> if forcing || isEmpty co || not (co &? o)
>                                     then Disappearing else Suspended o')
>   ,receive = (\ forcing co g @ (name,d,o') -> case co of
>       Just 0 -> (False, [forward g], null)
>       _ -> if name 'elem' targs
>           then case receive o forcing co g of
>               (False, gs, ol) -> (False, [forward g], null)
>               r -> r
>           else rcverror g "delay")
>   ,openTargets = []
>   ,delayedTargets = targs
>   ,height = 0
>   }
> targs = targets o
```

5 Extensions

If in our example document format there is a footnote on a page with only little other material, then this footnote will be appended immediately below that material instead of being pushed to the bottom of the page. Since we do not have anything corresponding to the gap modes of Lout, which influence the space behaviour of concatenation and can be used for such purposes, we might instead want to define

```
> vfill = recurse (prefix [""])

```

and insert this on top of `footSect` for pushing the footnote sections to the bottom of the page.

However, as mentioned before, this un-receptive recursion does not work well with our present evaluation strategy. Nevertheless we can introduce a combinator roughly corresponding to the Lout symbol `@VExpand` that makes an object take up all space available for it (as long as that space is finite):

```
> vExpand :: Object -> Object
> vExpand o = o' where
>   o' = Object
>     {eval = (\ forcing co -> case co of
>       Nothing ->      eval o forcing co
>       Just 0 ->      eval o forcing co
>       Just h -> case eval o forcing co of
>         Disappearing -> Yielding (strut h) null
>         NoSpace o1 -> NoSpace o1
>         Sending gs o1 -> Sending gs (vExpand o1)
>         Suspended o1 -> Suspended (vExpand o1)
>         Yielding c o1 -> Yielding c (if length c < h then vExpand o1 else o1))
>   ,receive = (\ frc co g -> thrdUpd3 vExpand (receive o frc co g))
>   ,openTargets = openTargets o
>   ,delayedTargets = delayedTargets o
>   ,height = height o
>   }
```

If we now modify the definition of page accordingly, then footnotes are always pushed to the bottom of the page:

```
> page = high 12 (vExpand (target "TextPlace") # footSect)

```

Another extension that we would like to present will allow us to number our pages.

We give a variant of the `recurse` combinator that uses functions from arbitrary domains to `Objects` instead of just `Objects`. The definition is unchanged except for the framework needed to set up the two expanded objects:

```
> recurseF :: ((a -> Object) -> (a -> Object)) -> (a -> Object)
> recurseF ff = f
>   where
>     f' = ff f
>     f a = o where
>       ffo = f' a
>       ff0 = ff (const null) a
>       targs = targets ffo
>       o = Object -- { as before }
```

The modifications necessary to obtain numbered pages are now very easy: the numbered page itself is obtained via a Haskell function taking the page number and delivering a numbered page (including the `vExpand` from above), and the list of numbered pages

is a recursion over a function that maps a page-list generator `mkpl` to another page-list generator that generates a numbered page concatenated with a recursive call using an incremented page number:

```
> npage :: Int -> Object
> npage n = high 14 $ prefix ["          - " ++ show n ++ "-",""]
>                (vExpand (target "TextPlace") # footSect)

> npageList :: Object
> npageList = let f mkpl n = npage n # mkpl (n+1)
>              in recurseF f 1
```

6 Concluding Remarks

Motivated by the galley concept implemented in Lout, we have presented Haskell definitions of eight elementary and two advanced document formatting combinators. These combinators allow to put together document formats that include page numbering, footnotes and tables of contents in a simple, declarative and intuitive way. We have not strived to mimick exactly the behaviour of Lout, but have instead tried to construct a self-contained, intuitively motivated and explainable implementation of the document formatting combinators. The definitions of those combinators, although they are not trivial and may not yet be the most elegant, still fitted into the space constraints set for this paper and show the elegance of Kingston’s galley abstraction and how this may be reflected in a purely functional formalisation.

Especially the extension to recurse over object-yielding functions also shows how document formatting can profit from being embedded into a full-fledged functional programming language, just as with many other domain-specific languages.

Although much still needs to be done to be able to reflect also the other abstractions underlying Lout in a Haskell setting, these combinators perhaps may serve to illuminate a little bit the path towards Jeff Kingston’s “vapourware successor to Lout”:

- A Haskell implementation of Lout or its successor would allow compiled document formats, so that only the document itself needed to be interpreted.
- Furthermore, document format authors could be given the choice whether to program in Lout or directly in Haskell, where the full power of Haskell would be at their fingertips.
- The concise functional specification could also serve as a more rigorous yet accessible documentation — the Lout mailing list abounds of postings by users baffled by the intricacies of the galley flushing algorithm.

An interesting application of the galley concept could also be found in literate programming: There variants of the same document parts are directed at different targets, namely program code files and documentation, very much like section headings are directed to their place above the section and to the table of contents.

Such an extension will be much easier to explore from within a Haskell setting than from the C implementation — one point to take care of is that we relied on the absence of side-effects in the definition of `recurse`, so that output to files either should not happen during `receives`, or the definition of `recurse` (and the forcing strategy) would have to be adapted.

Since the document formatting combinators presented in this paper are purely functional, and since their interaction is rather complicated, it is not easy to find the reasons for unexpected behaviour during development, especially since the relation between cause and effect may be very indirect and obscure. Since the author does not have access to any usable Haskell debugger or tracer, the Haskell definitions of this paper have been translated into term graph rewriting rules for the graphically interactive strongly typed second-order term graph transformation system HOPS [4,5]. Being able to *watch* things go wrong was invaluable for tracking down subtle design errors.

This paper has been typeset using the Lout document formatting system. Lout definitions and an auxiliary Haskell program provided by the author turn this paper into a literate program; the Haskell code generated from this paper with the use of Lout and FunnelWeb [12] is available on the WWW at

URL: <http://diogenes.informatik.unibw-muenchen.de/kahl/Haskell/VGalleys.html>

References

1. Pablo R. Azero, S. Doaitse Swierstra. Optimal Pretty-Printing Combinators, 1998. URL <http://www.cs.ruu.nl/groups/ST/Software/PP/>.
2. John Hughes. Pretty-printing: An Exercise in Functional Programming. In R. S. Bird, C. C. Morgan, J. C. P. Woodcock (eds.), *Mathematics of Program Construction*, pages 11–13. LNCS 669. Springer-Verlag, 1992.
3. John Hughes. The Design of a Pretty-printing Library. In J. Jeuring, E. Meijer (eds.), *Advanced Functional Programming*, pages 53–96. LNCS. Springer-Verlag, 1995.
4. Wolfram Kahl. The **H**igher **O**bject **P**rogramming System — User Manual for HOPS, Fakultät für Informatik, Universität der Bundeswehr München, February 1998. URL <http://diogenes.informatik.unibw-muenchen.de:8080/kahl/HOPS/>.
5. Wolfram Kahl. Internally Typed Second-Order Term Graphs. In J. Hromkovic, O. Sýkora (eds.), *Graph Theoretic Concepts in Computer Science, WG '98*, pages 149–163. LNCS 1517. Springer-Verlag, 1998.
6. Jeffrey H. Kingston. The design and implementation of the Lout document formatting language. *Software — Practice and Experience* **23**, 1001–1041 (1993).
7. Jeffrey H. Kingston. *An Expert's Guide to the Lout Document Formatting System (Version 3)*. Basser Department of Computer Science, University of Sydney, 1995.
8. Jeffrey H. Kingston. *A User's Guide to the Lout Document Formatting System (Version 3.12)*. Basser Department of Computer Science, University of Sydney. ISBN 0 86758 951 5, 1998. URL <ftp://ftp.cs.su.oz.au/jeff/lout>.
9. Donald E. Knuth. *The TEXBook*. Addison-Wesley, 1984.
10. Simon Peyton Jones. A Pretty Printer Library in Haskell, Version 3.0, 1997. URL <http://www.dcs.gla.ac.uk/~simonpj/pretty.html>.
11. Philip Wadler. A Prettier Printer, 1998. URL <http://cm.bell-labs.com/cm/cs/who/wadler/papers/prettier/>. Draft paper
12. Ross N. Williams. FunnelWeb User's Manual, May 1992. URL <http://www.ross.net/funnelweb/introduction.html>. Part of the FunnelWeb distribution

Lambda in Motion: Controlling Robots with Haskell

John Peterson¹, Paul Hudak¹, and Conal Elliott²

¹ Yale University, peterjohn@cs.yale.edu and paul.hudak@yale.edu

² Microsoft Research, conal@microsoft.com

Abstract. We present our experiences using a purely functional language, Haskell, in what has been traditionally the realm of low-level languages: robot control. *Frob* (Functional Robotics) is a domain-specific language embedded in Haskell for robot control. Frob is based on *Functional Reactive Programming* (FRP), as initially developed for Fran, a language of reactive animations. Frob presents the interaction between a robot and its stimuli, both onboard sensors and messages from other agents, in a purely functional manner. This serves as a basis for composable high level abstractions supporting complex control regimens in a concise and reusable manner.

1 Introduction

Robotics is an excellent problem domain to demonstrate the power and flexibility of declarative programming languages. Among the more interesting problem areas in this domain are control systems, constraint solving, reactive programming, sensor fusion, and real-time control. We have developed *Frob* (for Functional Robotics) as a domain-specific language, embedded in Haskell [PH97], for use in robotic systems. Frob hides the details of low-level robot operations and promotes a style of programming largely independent of the underlying hardware.

Our contributions include:

- Identification of factors that distinguish robotics programming from other uses of functional reactive programming.
- A characterization of certain aspects of robotics programming as *control system design*, and the use of continuous, mutually recursive equations in Frob to describe them.
- High-level abstractions that generalize patterns of control, including a monadic sequencing of control tasks.
- Combinators that intelligently fuse competing views of sensor data or strategies for robot control.

2 The Problem Domain

Our overall goal is to study control languages for general robotic systems. At present we have focused on a specific hardware configuration: a set of small

mobile robots communicating via a radio modem. These robots are Nomadics Superscouts controlled by an onboard Linux system. The robots contain three types of sensors: a belt of 16 sonars, bumpers for collision detection, and a video camera. The sonars give only an approximate idea of how close the robot is to an object: objects which are too short or do not reflect sonar are not detected. The drive mechanism uses two independent drive wheels and is controlled by setting the forward velocity and turn rate. The robot keeps track of its location via dead reckoning.

The robots are controlled by a Frob program running on top of Hugs, a small portable Haskell interpreter. Libraries supplied by Nomadics, written in C++, perform low-level control and sensor functions. These libraries are imported into Hugs using GreenCard II, a C/C++ interface language for Haskell. The running Frob program interacts with a remote console window via telnet, allowing the program to respond to keyboard input as well as its sensors. Robots may also send messages to each other via the radio modem, allowing us to explore cooperative behaviors.

In developing Frob we have relied on our experience working with *Fran*, a DSL embedded in Haskell for *functional reactive animation* [EH97, EI98b, EI98a]. Specifically, we have borrowed the core *behavior* and *reactivity* components of Fran, which together we call *FRP* (for *functional reactive programming*). However, FRP itself is not enough to deal with the additional complexities that arise in the realm of robotics (for one thing, an animated figure will always do what you ask; but a robot will not!). Amongst the distinguishing features of Frob are:

1. Robotics is characterized by multiple, tightly coupled closed-loop control systems. The equational and highly recursive nature of functional programming lends itself well to this task.
2. Robots live in a real world where errors of all sorts can occur almost anywhere at anytime. In Frob, we can build flexible error handling schemes directly into our abstractions.
3. An equally important kind of failure arises from unreliable/noisy data from sensors, requiring filters or correlation measures with other sensors. Frob must deal intelligently with uncertainty in its inputs.
4. Robots are full of gadgets that often undergo configuration changes. These devices may run at varying rates and require various degrees of responsiveness from the controller. This raises the need for flexibility, modularity, and platform independence in our software.
5. Frob programs must allow precise control over many “lower-level” aspects of the system such as sensor sampling rates and other hardware-level concerns.
6. Finally, there is a need for *planning and strategizing*. One of our long-term goals is to program robots with some reasonable level of intelligence. High-level abstractions in a functional programming setting should help us once again to step back from low-level details and concentrate on the big picture.

3 Frob and Functional Reactive Programming

FRP defines two essential representations used in Frob: *behaviors* and *events*. Behaviors are conceptually continuous quantities that vary over time, similar to the following type:

```
type Behavior a = Time -> a    -- simplified
```

For example, a value of type `Behavior SonarReading` represents a time-varying sonar reading, `Behavior Velocity` represents a time-varying robot velocity, and `Behavior Time` represents time itself.

Using behaviors to model robotic components hides unnecessary details that clutter the controller software. Continuous values may be combined without regard to the underlying sampling rate, eliminating some of the difficulties encountered in multi-rate systems. For example, consider using a robot-mounted camera to determine the position of an object. The tracking software yields a vector relative to the robot position; the absolute position of the tracked object combines this relative position with the current robot position and orientation. As continuous signals, these values can be added directly even when the camera is clocked at a different rate than positional samples.

Not all values are best represented in a continuous manner, however. For example, the robot bumpers generate discrete events rather than continuous values. Similarly, devices such as the keyboard are best kept in the discrete domain. This leads to Frob's notion of an *event*, which can be thought of as a stream of time/value pairs:

```
type Event a = [(Time,a)]    -- simplified
```

For example, an `Event BumperEvent` occurs when one of the robot bumper switches is activated, and an `Event Char` occurs when a console key is pressed.

The interaction between behaviors and events is the basis of Frob's notion of *reactivity*. Events and behaviors may change course in response to a stimulating event. Fran (and Frob) define a large library of useful built-in combinators, allowing complex behaviors to be described and combined in useful ways. For example, this function:

```
goAhead :: Robot -> Time -> WheelControl
goAhead r timeMax =
  forward 30 'untilB'
    (predicate (time > timeMax) .|.
     predicate (frontSonarB r < 20))) ==> stop
```

can be read, for robot `r`: “Move forward at a velocity of 30 cm/sec, until either time exceeds the limit, `timeMax`, or an object appears closer than 20 cm, at which point stop.” The operations used here are typical FRP primitives:

- `untilB` changes behavior in response to an event,
- `predicate` generates an event when a condition becomes true,

- `.|. .` interleaves two events streams, and
- `-->` associates a new value (in this case the behavior `stop`) with an event.

Although the interaction between Frob and the robot sensors is naturally implemented in terms of events (discrete samplings of sensor values), it is convenient to smooth out these values into a continuous behavior by extrapolating the discrete readings. Conversely, robot controls defined in a continuous manner must be sampled into discrete messages to the robot. It is generally more convenient to deal with continuous behaviors rather than discrete events. Frob generally provides both views of input sensors, both discrete and continuous, to the programmer, providing a choice of the most appropriate representation.

3.1 A Wall Follower

We now turn to a simple example of a robot control program: wall following. In this example, two sensor readings guide the robot: a front sensor determines the distance to any obstacle in front of the robot, and a side sensor measures the distance to the wall. In addition, the motor controller provides the actual robot velocity (this may differ from the velocity requested of the motors). Domain engineers typically describe such a task using differential equations. The equations describing this controller are:

$$v = \sigma(v_{\max}, f - d^*)$$

$$\omega = \sigma(\sin(\theta_{\max}) * v_{\text{curr}}, s - d^*) - \dot{s}$$

where $\sigma(x, y)$ is the limiting function $\sigma(x, y) = \max(-x, \min(x, y))$, d^* is the desired “setpoint” distance for objects to the front or side of the robot, v_{\max} and θ_{\max} are the maximum robot velocity and body angle to the wall, respectively, and \dot{s} denotes the derivative of s . In Frob, these equations become:

```
type FloatB          = Behavior Float
type WheelControl = Behavior (Float, Float)

basicWallFollower ::
  FloatB -> FloatB -> FloatB -> FloatB -> WheelControl

basicWallFollower vel side front setpoint =
  pairB v omega -- creates a behavior of tuples
  where
    v          = limit vmax (front - setpoint)
    omega      = targetSideV - derivative side
    targetSideV = limit (vel * sin maxAngleToWall)
                  (setpoint - side)

limit high x = (-high) 'max' x 'min' high
```


The type `WheelControl` defines the output of this system, a pair of numbers defining the forward velocity and turn rate of the robot. Thus `basicWallFollower` is a direct encoding of the control equations, with `front` and `side` as the sonar readings, `vel` is the robot velocity, and `setpoint` as the distance the robot attempts to maintain from the wall.

The strategy expressed here is fairly simple: the robot travels at a maximum velocity until blocked in front, at which time it slows down as it approaches an obstacle. The value `targetSideV` is the rate that the robot should ideally approach the wall at; note this will be zero when `side` matches `setpoint`. This is limited by `maxAngleToWall` to prevent the robot from turning too sharply toward or away from the wall. The rotation rate, `omega`, is determined by the difference between the desired approach rate and the actual approach rate, as measured by taking the derivative of the side sonar reading.

It is important to emphasize the declarative nature of this short Frob program; it is essentially a rewrite of the differential equation specification into Haskell syntax. There is no loop to iteratively sample the sensors, compute parameters, update control registers, etc. In general, details pertaining to the flow of time are hidden from the programmer. Expressions such as `targetSideV - derivative side` represent operations on behaviors; overloading permits us to write them clearly and succinctly. Some operators, notably `derivative` in this example, directly exploit the time-varying nature of the signals. Finally, this code is independent of the kind of sensors used to measure the distances.

We demonstrate the use of reactivity by extending this example to pass control to a new behavior, represented as a continuation, when either the end of the side wall is encountered or an obstacle blocks the robot. A boolean value is passed into the continuation to distinguish the two cases.

```

wallFollower :: (Bool -> WheelControl) ->
  FloatB -> FloatB -> FloatB -> FloatB -> WheelControl
wallFollower cont side front rate dist =
  basicWallFollower side front rate dist 'untilB'
    ((predicate (side > sideThreshold)) ==> cont True ) .|.
    ((predicate (front < frontThreshold)) ==> cont False)

```

This can be read: “Follow the wall until the side reading is greater than its threshold or the front reading is less than its threshold, then follow behavior `cont`.”

4 More Abstractions

So far, we have demonstrated the ability of FRP to directly encode control equations and have used basic reactivity (`untilB`) to shape the overall system behavior. However, the real power of functional programming lies in the ability to define new, higher-level abstractions based on these primitive operators. Since Frob is embedded in Haskell, the full power of higher-order functional programming is available to the user. A number of useful abstractions are also pre-defined by Frob. In this section we examine some of these “robotic building blocks”.

4.1 A Task Monad

Although FRP’s reactivity is capable of describing arbitrary reactive behaviors, it is useful to add a layer on top of the basic FRP operations to lend more structure to the system. Indeed, the basic `untilB` operator is somewhat like a general “goto” construct: fundamental yet easily abused. It is often natural to think in terms of sequential *tasks*. The `wallfollower` function defined in Sect. 3 is an example of this: once the wall following task is complete, it calls a continuation to initiate the next task. We can alternatively use a *monad* to express sequential task combination. We also incorporate a number of other useful features into this monad, including:

- Exception handling.
- Implicit capture of task start time and robot state, allowing the task to be defined relative to the time or robot position at the start of the task. Tasks such as “wait 10 seconds” or “turn 90 degrees right” require this initial state information.
- Implicit propagation of the associated robot definition from task to task.
- A form of interrupt monitoring that adds interrupt events to a given task.

This monad hides the details of task sequencing, simplifying programming. Tasks are represented by the following types:

```
type TState = (Robot, RState, Event RoboErr)
type RState = (Time, Point2, Angle)
data Task b e =
  Task (TState -> (TState -> Either RoboErr e -> b) -> b)
```

The type `b` is the behavior defined by the task; `e` is the type of value returned at task completion. The task state, `TState`, contains the task’s associated robot, the state of the robot at the start of the task, and a locally scoped error-catching event. The robot state, of type `RState`, saves the time, location, and heading of the robot at the start of the task. We use standard monadic constructions combining a state monad and an exception monad; the monad instance for `Task a` is straightforward:

```
instance Monad (Task a) where
  Task t1 >>= u =
    Task (\ts cont ->
      t1 ts (\ts' res ->
        case res of
          Right v -> case u v of
            Task t2 -> t2 ts' cont
          Left l -> cont ts' (Left l)))
  return k = Task (\ts cont -> cont ts (Right k))
```

The state is managed in the underlying task functions. These are created by the `mkTask` function using a triple consisting of

1. the behavior during the task,
2. an event defining successful completion of the task, and
3. another event used to raise an error, if any.

Specifically:

```
mkTask :: (Robot -> RState ->
           (Behavior a, Event b, Event RoboErr))
        -> Task a b

mkTask f =
  Task (\(r,rstate,err) nextTask ->
        case f r rstate of
          (beh, success, failure) ->
            beh 'untilB'
              ((success ==> Right) .|.
               (failure .|. err) ==> Left)
              'snapshot' stateOf r
              ==> \(res, rs') -> nextTask (r,rs',err) res)

stateOf r = pairB time (pairB (positionB r) (orientationB r))
```

This function defines the basic structure of a task: a behavior which continues until either a termination event or an error event stops it. The `snapshot` function captures the state of the robot, passing it into the next task. The termination event managed by the monad, `err`, supplies error handling from outside the context of the current task. This allows construction of composite tasks which share a common error exit; while `failure` defines an error event specific to a single task, the monad allows failure events to scope over larger sets of tasks.

These functions are the basic task combinators; their purpose is generally obvious from the type signature:

```
taskCatch      :: Task a b -> (RoboErr -> Task a b) -> Task a b
taskError      :: RoboErr -> Task a b  -- Raise an error

withTaskError  :: Task a b -> Event RoboErr -> Task a b
timeLimit      :: Task a b -> Time -> b -> Task a b

getStartTime   :: Task a Time
getInitialPos  :: Task Point2
getRobot       :: Task a Robot

runTask        :: Robot -> RState -> Task a b -> Behavior a
                -- Generate an error if the task terminates
```

The `withTaskError` and `timeLimit` functions use locally scoped error handlers (part of the `TState` type) to add termination conditions to an existing task, either with an arbitrary error event (`withTaskError`) or by aborting the task with a specified return value if it does not complete in a specified time period.

The `runTask` function converts a task into an ordinary behavior, aborting if the task terminates.

Using task functions, we now define interesting, easily understood robot control primitives represented as tasks and sequenced using Haskell's `do` notation. For example, using the `mkTask` function, the wall follower of Sect. 3.1 may be defined as follows:

```

wallFollower1 :: FloatB -> FloatB -> FloatB ->
               FloatB -> Task RControl Bool
wallFollower1 side front rate d =
  mkTask
    (\r _ -> (wallFollower front side d rate,
               predicate (side > sideThreshold) ==> True |.
               predicate (front < frontThreshold) ==> False,
               crash r ==> HitBumper))

```

This adds a new feature to the wall follower: if the robot's bumper, represented by the `crash` event, is triggered the error handler is called. The value `HitBumper` is part of the `RoboErr` type and tells the error handler the source of the error.

The use of a task monad does not imply single-threadedness. Tasks simply define behaviors; different tasks may be running simultaneously to control different parts of the robot. Tasks may be combined in the same manner that other behaviors are combined.

Here we present a few other common operations encoded as tasks. This turns the robot away from an obstacle:

```

turnAway :: Task WheelControl ()
turnAway = mkTask (\r _ -> (constantB (0, 0.1),
                             predicate (frontSonar r >= 30),
                             neverE))

```

The robot turns at a constant rate until the front sonar indicates that there is at least 30cm of open space in front of the robot.

This task turns 90 degrees:

```

turnRight :: Task WheelControl ()
turnRight = mkTask (\r rstate ->
  let target = constantB (orient rstate +@ (90 # deg)) in
  (constantB (0, 0.1),
   predicate (orientationB r === target),
   neverE))

```

The `===` operator is “nearly equal” comparison; exact comparison with the target orientation is not likely to succeed. The `+@` operation is angle addition; the result is normalized to the 0 to 2π range.

This task drives the robot to a specific location. We start with a behavior to drive the wheels:

```

goToward :: Point2B -> Robot -> Behavior RControl
goToward p r = pairB vel ang
  where
    err          = p .-. robotPosB r -- vector to target
    (delta, a)   = vector2PolarCoords err -- changed to polar
    da          = a -@ robotHeadingB r
    vel          = limit (maxVelocity r) (k1 * delta) -- velocity
    ang          = limit (maxTurn r) (k2 * da)         -- turn rate

```

Again, this program corresponds closely to the equations a domain engineer would use to specify this system. Adding an event that detects arrival (within a suitable radius) yields a task:

```

travelTo :: Point2B -> Task RControl ()
travelTo p = mkTask (\r -> goToward p r,
                    predicate (dist p r < howClose),
                    senseBlocked r ==> Blocked)

```

This definition chooses (somewhat arbitrarily) to call the error handler when the sonars indicate obstacles ahead of the robot.

Finally, this function iterates a task until successful completion, using an error handler that restarts it repeatedly on failure.

```

keepTrying :: Task a b -> Task a c -> Task a b
keepTrying t err = t 'taskCatch'
                  \_ -> do err; keepTrying t err

```

We use this strategy to combine to previously defined tasks into a composite behavior:

```
goto place = keepTrying (travelTo place) turnAway
```

As defined earlier, `travelTo` moves toward a specific location, terminating normally on arrival but raising an error if the robot is blocked. In this example, each time the robot ‘steps aside’ in `turnAway`, it tries again once the obstacle is out of the way.

4.2 Fusion

A common idiom in the robotics world is *fusion*. In general, fusion is the integration of routines which either observe similar parts of the real world or control the same (or similar) parts of the robot behavior. Some examples are:

- Combining sensor information into a unified picture of the outside world. For example, a robot with both vision and sonar has two different views of the same environment. Combining these views into a unified picture of the outside world is a crucial aspect of robotic control.

- Resolving conflicting doctrines. For example, an overall goal of traveling to a specific destination may conflict obstacles in the shortest path. The obstacle avoider may present a set of alternative paths; knowledge of the overall goal is needed to choose the route likely to be the best.
- Combining the behavior of more than one robot into an overall coordinated behavior.

Frob supports simple, reusable definitions of general fusion strategies.

One sort of fusion blends competing goals rather than selecting a current goal of attention. Perhaps the simplest example of this uses *force vectors*: imaginary forces pushing on the robot. These forces combine through vector addition. The following code assigns a force to a sonar reading:

```
sonarforce :: Robot -> SonarReading -> Vector2
sonarforce robot sonars =
  sum (zipWith f sonar (sonarAngles r) sonars)
  where
    f reading angle = vector2polar (distToForce reading) angle
    distToForce d | d >= maxSonar r = 0
                  | d < minValidSonar = 0
                  | otherwise = (-1) / (d*d)
```

This generates a force that increases as the sonar distance decreases and points in a direction opposite of the angle of the sonar (hence the `-1`). This next function generates a force pushing towards a goal, limited by a maximum force:

```
forceToGoal :: Point2 -> Point2 -> Vector2
forceToGoal there here = f where
  err      = there .-. here
  (power, a) = vector2PolarCoords err -- changed to polar
  f        = vector2polar (limit power maxForce) a
```

Finally, this defines travel to destination, this time with obstacle avoidance:

```
goTowardF :: Point2B -> Robot -> Behavior RControl
goTowardF p r = followForce f r
  where f = lift1 (sonarForce r) (sonarB r) +
          (lift2 forceToGoal) p (positionB r)
```

The `goTowardF` function combines forces generated by obstacles on the sonars and a pull toward the destination to yield a simple controller that blends the competing goals of obstacle avoidance and travel to a specific place. The `lift` functions are necessary to convert scalar functions such as `sonarForce` into operations on behaviors. Not shown is the function `followForce`; it drives the robot based on the direction of force.

Sensor fusion is also neatly expressible in Frob. Consider an example taken from a robotic soccer competition. A team of vision-equipped robots tracks the soccer ball. The tracking software on each robot determines both the location of

the ball when visible (based on the angle and the apparent size of the ball) and also assigns a confidence value to the reading, based on how sure the tracker is of the ball position. Each robot broadcasts the output of its tracker and these differing views of the ball are fused. This code generalizes this concept:

```
type WithConfidence a = (Float, a)
type WithConfidenceB a = Behavior (Float, a)
mergeB :: WithConfidenceB a -> WithConfidenceB a ->
        WithConfidenceB a
trackAll :: [Robot] -> WithConfidenceB Point2
trackerAll team = foldr1 mergeB (map myTracker team)
```

We use the behavioral form of `withConfidence` to create a confidence varying over time. We decay confidence values, preferring recent low confidence readings over older high confidence readings. The following sampling code converts events (sensor readings) into a continuous behavior with linearly decaying confidence:

```
eventToBehaviorC :: Event (Float,a) -> a -> WithConfidenceB a
eventToBehaviorC e init =
    switcher (constantB (0, init)) e'
  where
    e' = e 'withTimeE' \(eventTime, (c,val)) ->
        pairB (c * decay eventTime) (constantB val)
    decay t0 = (1 - (time - constantB t0) / decayInterval)
              'max' 0
```

The `switcher` function assembles a behavior piecewise, with each event defining a new segment of the behavior. Initially the behavior is undefined until the first sensor reading arrives. At each occurrence, a pair containing the sampled value, a constant, and its confidence, a linearly decaying value, is generated.

5 Implementing Frob

Frob is built by adding two components to the basic FRP engine, used unaltered from the Fran distribution. Underneath FRP is a layer that mediates between the IO actions that interact directly with the robot and the events native to FRP. On top of FRP are the various Frob abstractions for robotics, including coercions between the event streams generated by sampling and continuous behaviors.

Communication from FRP to the outside world is handled by event “listeners”. A listener is a function that gets invoked on every occurrence of an event to which it is attached. That is, given an event of type `Event a`, its listeners are Haskell functions of type `a -> IO ()`. This mechanism is how information is moved out of the FRP domain into the robot.

Another mechanism, called “repeaters”, supports getting information from the outside world *into* the FRP domain. A repeater is just a linked pair of listener and event. The event repeats whatever the listener is told. Creation of an FRP event based on some external real-world event is done by invoking `newRepeater`

of type `IO (Listener a, Event a)` to get a pair `(l,e)`. Then whenever the external event occurs, imperative code tells the listener `l`, which causes the FRP event `e` to occur.

During initialization, event streams for all input sources are created using `newRepeater`. Listeners are attached to the output events of the user's program to forward each external event of interest. New repeaters and listeners may also be created on the fly, allowing new event streams to appear as the program executes.

Finally, the issue of *clocking* must be addressed. The timing of the input events (repeaters) defines the clocking of the system. There are no apriori constraints on this clocking: a repeater may be called at any time or rate; one sensor may be sampled frequently and another only occasionally. Input events may be driven by either interrupts or by polling. Once an event is injected into the system by a repeater, all computation contingent on that event is immediately performed, perhaps resulting in calls to repeaters.

At present, we use the simplest possible clocking scheme: a single heartbeat loop that runs as fast as possible, sampling all inputs and pushing their values into the system. The speed of the heartbeat loop depends on how long it takes to handle the incoming events. This is not ideal; if the heartbeat goes too slow the responsiveness of the system suffers. In the future we hope to explore more complex clocking schemes.

The type `Robot` is a container for all of the robot state, both dynamic (the events and behaviors defined by its sensors) and static (the description of hardware configuration). The dynamic values are generally provided in two forms: as the raw events generated by the hardware and as continuous behaviors. This "smoothing" of the event streams into a continuous form allows the sensors to be represented in a convenient way. A typical definition of `Robot` is:

```
data Robot = Robot {position      :: Event Point2,
                    positionB    :: Behavior Point2,
                    orientation   :: Event Float,
                    orientationB  :: Behavior Float,
                    sonar         :: Event SonarReading
                    sonarB        :: Behavior SonarReading
                    maxVelocity   :: Float,
                    sonarAngles   :: [Float],
                    ...}
```

Similarly, the type `RobotControl` encapsulates all system outputs. The wheel controller, for example, is contained inside this structure. Output sampling is similar to input smoothing; here continuous behaviors are sampled (using the `snapshot` function) into discrete events. Both smoothing and sampling take place within the purely functional domain of FRP; only the listeners and talkers are relegated to the imperative world.

This code sketches the basic structure of Frob's main loop:

```
runRobot :: (Robot -> RobotControl) -> IO ()
runRobot f = do
    -- Create repeaters for all input values
    (inputEv1, listener1) <- newRepeater
    ...
    -- Combine all inputs into type "Robot"
    let robot = addSmoothing (makeRobot inputEv1 ...)
        -- Define output behaviors
        control = f robot
        -- convert to events
        (outputEvent1,...) = sampleBehaviors control
    -- Set up listeners for all output events
    addListener outputEvent1 (\t value -> ...)
    ...
    -- enter the heartbeat loop
    loop
where loop =
    do -- for each repeater, get robot data and post to event
        i1 <- getRobot1
        listener1 i1
        ...
        loop
```

6 Experience

We have built a number of small controllers using Frob. Our experiences to date with the practical side of the system include:

- It was easy to use functional reactive programming to model our interaction with the outside world. Packaging up the robot controls as FRP events was simple and resulted in a much more declarative programming style than would be possible with an imperative language.
- Frob programs are far smaller than corresponding C++ programs for the relatively small systems developed so far. Abstractions such as tasks are much easier to implement and use in Frob than in C++. Experience with computer vision applications suggests that these same abstractions can be directly encoded in C++ (making judicious use of templates to handle polymorphism) but this requires considerably more effort and in most cases is not the obvious implementation choice in C++.
- The experimental nature of our robot control applications makes rapid prototyping particularly essential. New behaviors can be written quickly and succinctly in Frob, allowing the developer to spend more time actually experimenting with the robot.

- Performance has not been a problem, in spite of the fact that we are running Haskell using an interpreter (Hugs). The feedback control systems driving the robot are very robust and slow clock rates or garbage collection delays do not perturb the system enough to cause any real problems.

7 Related Work

Several researchers have found declarative languages well suited for modeling pictures, 3D models, and even music. Examples include [Hen82, LZ87, Bar91, ZLL⁺88, FJ95, HMGW96]. Arya used a functional language to model 2D animations as lazy lists of pictures constructed using list combinators [Ary94]. While this work was quite elegant, the use of lists implies a discrete model of time, which is somewhat unnatural. The TBAG system modeled 3D animations as functions over continuous time, using a “behavior” type family [ESYAE94]. Although behaviors were based on continuous time, reactivity was handled imperatively through constraint assertion and retraction. Fran and DirectAnimation both grew out of the ideas in an earlier design called ActiveVRML [Ell96].

There has also been related work on concurrency and reactivity. CML (Concurrent ML) formalized synchronous operations as first-class, purely functional, values called “events” [Rep91]. In CML, events are ultimately used to perform an action, such as reading input from or writing output to a file or another process. In contrast, our events are used purely for the values they generate. Concurrent Haskell [JGF86] extends Haskell with a small set of primitives for explicit concurrency, designed around monadic I/O. While this system is purely functional in the technical sense, its semantics has a strongly imperative feel. In contrast, modeling entire behaviors as implicitly concurrent functions of continuous time yields what we consider a more declarative feel.

8 Conclusions

Assessing the effectiveness of Frob in the realm of robotics, it has succeeded in a number of ways:

- The basic building blocks of robotic systems such as differential equations, reactivity, and state machines, are all directly supported by Frob, allowing systems to be programmed directly from their specification.
- FRP effectively hides the imperative nature of the underlying robotic system, supporting a declarative programming style.
- The low-level details of the robotic system have been hidden under abstractions, allowing programming in a manner that is largely independent of the specific underlying hardware.
- We have been able to define a number of general control strategies in a modular and reusable way.

We are now in the process of incorporating vision-based controllers into our framework. We plan to use Frob for more complex systems soon. The latest release of our system is available on the web at haskell.org/frob.

9 Acknowledgements

This research was supported by NSF Experimental Software Systems grant CCR-9706747.

References

- [Ary94] K. Arya. A functional animation starter-kit. *Journal of Functional Programming*, 4(1):1–18, 1994.
- [Bar91] Joel F. Bartlett. Don't fidget with widgets, draw! Technical Report 6, DEC Western Digital Laboratory, May 1991.
- [EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, pages 163–173, June 1997.
- [Ell96] Conal Elliott. A brief introduction to ActiveVRML. Technical Report MSR-TR-96-05, Microsoft Research, 1996.
- [Ell98a] Conal Elliott. Composing reactive animations. *Dr. Dobb's Journal*, July 1998. Extended version with animations at <http://research.microsoft.com/~conal/fran/{tutorial.htm,tutorialArticle.zip}>.
- [Ell98b] Conal Elliott. Fran user's manual. <http://research.microsoft.com/~conal/Fran/UsersMan.htm>, July 1998.
- [ESYAE94] Conal Elliott, Greg Schechter, Ricky Yeung, and Salim Abi-Ezzi. TBAG: A high level framework for interactive, animated 3D graphics applications. In *Proceedings of SIGGRAPH '94*, pages 421–434. ACM SIGGRAPH, July 1994.
- [FJ95] Sigbjorn Finne and Simon Peyton Jones. Pictures: A simple structured graphics model. In *Proceedings of Glasgow Functional Programming Workshop*, July 1995.
- [Hen82] P. Henderson. Functional geometry. In *Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 179–187. ACM, 1982.
- [HMGW96] P. Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3):465–483, May 1996.
- [JGF86] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *Proceedings of ACM Symposium on Principles of Programming Languages*. ACM SIGPLAN, January 1986.
- [LZ87] Peter Lucas and Stephen N. Zilles. Graphics in an applicative context. Technical report, IBM Almaden Research Center, July 1987.
- [PH97] John Peterson and Kevin Hammond. Haskell 1.4: A non-strict, purely functional language. Technical Report YALEU/DCS/RR-1106, Department of Computer Science, Yale University, May 1997.
- [Rep91] John H. Reppy. CML: A higher-order concurrent language. In *Conference on Programming Language Design and Implementation*, pages 293–305. SIGPLAN, June 1991.
- [ZLL⁺88] S.N. Zilles, P. Lucas, T.M. Linden, J.B. Lotspiech, and A.R. Harbury. The Escher document imaging model. In *Proceedings of the ACM Conference on Document Processing Systems*, pages 159–168, December 1988.

CHAT: The Copy-Hybrid Approach to Tabling

Bart Demoen and Konstantinos Sagonas

Department of Computer Science
Katholieke Universiteit Leuven
B-3001 Heverlee, Belgium
`{bmd,kostis}@cs.kuleuven.ac.be`

Abstract. The Copying Approach to Tabling, abbrev. CAT, is an alternative to SLG-WAM and based on total copying of the areas that the SLG-WAM freezes to preserve execution states of suspended computations. The disadvantage of CAT as pointed out in a previous paper is that in the worst case, CAT must copy so much that it becomes arbitrarily worse than the SLG-WAM. Remedies to this problem have been studied, but a completely satisfactory solution has not emerged. Here, a hybrid approach is presented: CHAT. Its design was guided by the requirement that for non-tabled (i.e. Prolog) execution no changes to the underlying WAM engine need to be made. CHAT combines certain features of the SLG-WAM with features of CAT, but also introduces a technique for freezing WAM stacks without the use of the SLG-WAM's freeze registers that is of independent interest. Empirical results indicate that CHAT is a better choice for implementing the control of tabling than SLG-WAM or CAT. However, programs with arbitrarily worse behaviour exist.

1 Introduction

In [2], we developed a new approach to the implementation of the suspend/resume mechanism that tabling needs: CAT. The essential characteristic of the approach is that freezing of the stacks (as in SLG-WAM [4]) was replaced by copying the state of suspended computations. One advantage is that this approach to implementing tabling does not introduce new registers, complicated trail or other inefficiencies in an existing WAM: CAT does not interfere at all with Prolog execution. Another advantage is that CAT can perform completion and space reclamation in a non-stack based manner without need for memory compaction. Finally, experimentation with new scheduling strategies seems more easy within CAT. On the whole, CAT is also easier to understand than the SLG-WAM. The main drawback of CAT, as pointed out in [2], is that its worst case performance renders it arbitrarily worse than the SLG-WAM: CAT might need to copy arbitrary large parts of the stacks; the SLG-WAM's way of freezing in contrast is an operation with constant cost. Although this bad behaviour of CAT has not shown up as a real problem in our uses of tabling (see [2] and the performance section of this paper), in [3] we have described a partial remedy for this situation. Restricted to the heap, it consists of performing a minor garbage

collection while copying; that is, preserve only the useful state of the computation by copying just the data that are used in the forward continuation of each consumer (a goal which resolves against answers from the table). The same idea can be applied to the local (environment) stack as well. [3] contains some experimental data which show that this technique is quite effective at reducing the amount of copying in CAT. This is especially important in applications which consist of a lot of Prolog computation and few consumers. However, even this memory-optimised version of CAT suffers from the same worst case behaviour compared to SLG-WAM. Nevertheless, for most applications CAT is still a viable alternative to SLG-WAM.

We therefore felt the need to reconsider the underlying ideas of CAT and SLG-WAM once more. In doing so, it became quite clear that all sorts of hybrid methods are also possible, e.g. one could copy the local stack while freezing the heap, trail and choice point stack, etc. However, we are convinced that the guiding principle behind any successful design of a (WAM-based) tabling implementation must be that the necessary extensions to support tabling should not impair the efficiency of the underlying abstract machine for (strictly) non-tabled execution, and support for tabled evaluation should be possible without requiring difficult changes: CAT was inspired by this principle and provides such a design. CHAT, the hybrid CAT we present here enjoys the same property.

If the introduction of tabling must allow the underlying abstract machine to execute Prolog code at its usual speed, we have to preserve and reconstruct execution environments of suspended computations without using SLG-WAM's machinery; in other words we have to get rid of the freeze registers and the forward trail (with back pointers as in the SLG-WAM). The SLG-WAM has freeze registers for heap, trail, local and choice point stack. These are also the four areas which CAT selectively copies. What CHAT does with each of these four areas is described in Section 3 which is the main section of this paper. Section 4 shows best and worst cases for CHAT compared to the SLG-WAM. Section 5 discusses the combinations possible between CHAT, CAT and SLG-WAM. Section 7 shows the results of some empirical tests with our implementation of CHAT and Section 8 concludes.

2 Notation and Terminology

Due to space limitations we assume familiarity with the WAM (see e.g. [1, 6]), SLG-WAM [4] and CAT [2]. We also assume a four stack WAM, i.e. an implementation with separate stacks for the choice points and the environments as in SICStus Prolog or in XSB. This is by no means essential to the paper and whenever appropriate we mention the necessary modifications of CHAT for the original WAM design. We will also assume stacks to grow downwards; i.e. higher in the stack means older, lower in the stack (or more recent) means younger.

We will use the following notation: **H** for top of heap pointer; **TR** for top of trail pointer; **E** for current environment pointer; **EB** for top of local stack pointer; **B** for most recent choice point; the (relevant for this paper) fields of a choice point are ALT, H and EB, the next alternative, the top of the heap and

local stack respectively at the moment of the creation of the choice point; for a choice point of type T pointed by \mathbf{B} , these fields are denoted as $\mathbf{B}_T[\text{ALT}]$, $\mathbf{B}_T[\text{H}]$ and $\mathbf{B}_T[\text{EB}]$ — T is either Generator, Consumer or Prolog choice point. The SLG-WAM uses four more registers for freezing the WAM stacks; however only two of them are relevant for this paper. We denote them by \mathbf{HF} for freezing the heap, and \mathbf{EF} for freezing the local stack.

In a tabling implementation, some predicates are designated as *tabled* by means of a declaration; all other predicates are *non-tabled* and are evaluated as in Prolog. The first occurrence of a tabled subgoal is termed a *generator* and uses resolution against the program clauses to derive answers for the subgoal. These answers are recorded in the table (for this subgoal). All other occurrences of identical (e.g. up to variance) subgoals are called *consumers* as they do not use the program clauses for deriving answers but they consume answers from this table. Implementation of tabling is complicated by the fact that execution environments of consumers need to be retained until they have consumed all answers that the table associated with the generator will ever contain.

To partly simplify and optimize tabled execution, implementations of tabling try to determine *completion* of (generator) subgoals: i.e. when the evaluation has produced all their answers. Doing so, involves examining dependencies between subgoals and usually interacts with consumption of answers by consumers. The SLG-WAM has a particular stack-based way of determining completion which is based on maintaining *scheduling components*; that is, sets of subgoals which are possibly inter-dependent. A scheduling component is uniquely determined by its *leader*: a (generator) subgoal G_L with the property that subgoals younger than G_L may depend on G_L , but G_L depends on no subgoal older than itself. Obviously, leaders are not known beforehand and they might change in the course of a tabled evaluation. How leaders are maintained is an orthogonal issue beyond the scope of this paper; see [4] for more details. However, we note that besides determining completion, leaders of a scheduling component are usually responsible for scheduling consumers of all subgoals that they lead to consume their answers.

3 The Anatomy of CHAT

We describe the actions of CHAT by means of an example. Consider the following state of a WAM-based abstract machine for tabled evaluation. A generator G has already been encountered and a *generator choice point* has been created for it immediately below a (Prolog) choice point P_0 ; then execution continued with some other non-tabled code (P and all choice points shown by dots in Figure 1). Eventually a consumer C was encountered and let us, without loss of generality, assume that G is its generator and G is not completed¹. Thus, a *consumer choice point* is created for C ; see Figure 1. The heap and the trail are shown segmented according to the values saved in the corresponding fields of choice points. The

¹ Otherwise, if G is completed, the whole issue is trivial as a *completed table optimization* can be performed and execution proceeds as in Prolog; see [4].

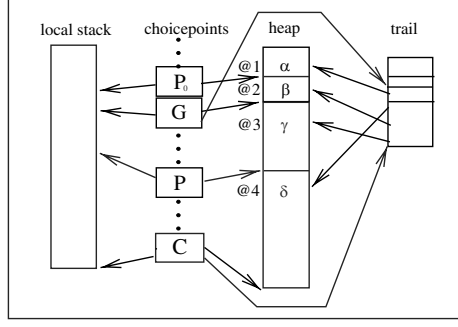


Fig. 1. CHAT stacks immediately upon laying down a consumer choice point.

same segmentation is not shown for the local stack as it is a spaghetti stack; however the EB values of choice points are also shown by pointers.

Without loss of generality, let us assume that C is the only consumer. The whole issue is how to preserve the execution environment of C . CAT does this very simply through (selectively and incrementally) copying all necessary information in a separately allocated memory area — see [2]. The SLG-WAM employs *freeze registers* and freezes the stacks at their current top; allocation of new information occurs below these freeze points — see [4]. We next describe what CHAT does.

3.1 Freezing the Heap without a Heap Freeze Register

As mentioned, we want to prevent that on backtracking to a choice point P that lies between the consumer C and the nearest generator G (included), \mathbf{H} is reset to the $\mathbf{B}_P[\mathbf{H}]$ as it was on creating P . However, the WAM sets:

$$\mathbf{H} := \mathbf{B}_P[\mathbf{H}]$$

upon backtracking to a choice point pointed to by \mathbf{B}_P . We can achieve that no heap lower than $\mathbf{B}_C[\mathbf{H}]$ is reclaimed on backtracking to P , by manipulating its $\mathbf{B}_P[\mathbf{H}]$ field, i.e. by setting:

$$\mathbf{B}_P[\mathbf{H}] := \mathbf{B}_C[\mathbf{H}]$$

at the moment of backtracking out of the consumer. Note that rather than waiting for execution to backtrack out of the consumer choice point, this can happen immediately upon encountering the consumer (see also [4] on why this is correct).

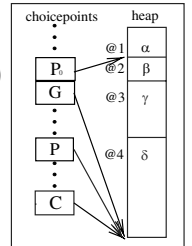
More precisely, upon creating a consumer point for a consumer C the action of CHAT is:

for all choice points P between C and its generator (included)

$$\mathbf{B}_P[\mathbf{H}] := \mathbf{B}_C[\mathbf{H}]$$

The picture on the right shows which \mathbf{H} fields of choice points are adapted by CHAT in our running example.

To see why this action of CHAT is correct, compare it with how the SLG-WAM freezes the heap using the freeze



register **HF**:

when a consumer is encountered, the SLG-WAM sets $\mathbf{HF} := \mathbf{B}_C[\mathbf{H}]$
 on backtracking to a choice point P , the SLG-WAM resets \mathbf{H} as follows:
 if $\text{older}(\mathbf{B}_P[\mathbf{H}], \mathbf{HF})$ then $\mathbf{H} := \mathbf{HF}$ else $\mathbf{H} := \mathbf{B}_P[\mathbf{H}]$

In this way, CHAT neither needs the freeze register **HF** of the SLG-WAM, nor uses copying for part of the heap as CAT.

The cost of setting the $\mathbf{B}[\mathbf{H}]$ fields by CHAT is linear in the number of choice points between the consumer and the generator up to which it is performed. In principle this is unbounded, so the act of freezing in CHAT can be arbitrarily more costly than in SLG-WAM. However, our experience with CHAT is that this is not a problem in practice; see the experimental results of Section 7

3.2 Freezing the Local Stack without **EF**

The above mechanism can also be used for the top of the local stack. Similar to what happens for the \mathbf{H} fields, CHAT sets the \mathbf{EB} fields in affected choice points to $\mathbf{B}_C[\mathbf{EB}]$. In other words, the action of CHAT is:

for all choice points P between the consumer C and its generator (included)
 $\mathbf{B}_P[\mathbf{EB}] := \mathbf{B}_C[\mathbf{EB}]$

The top of the local stack can now be computed as in the WAM:

if $\text{older}(\mathbf{B}[\mathbf{EB}], \mathbf{E})$ then $\mathbf{E} + \text{length}(\text{environment})$ else $\mathbf{B}[\mathbf{EB}]$

and no change to the underlying WAM is needed.

Again, we look at how the SLG-WAM employs a freeze register **EF** to achieve freezing of the local stack: **EF** is set to **EB** on freezing a consumer. Whenever the first free entry on the local stack is needed, e.g. on backtracking to a choice point \mathbf{B} , this entry is determined as follows:

if $\text{older}(\mathbf{B}[\mathbf{EB}], \mathbf{EF})$ then \mathbf{EF} else $\mathbf{B}[\mathbf{EB}]$

The code for the `allocate` instruction is slightly more complicated as a three-way comparison between $\mathbf{B}[\mathbf{EB}]$, **EF** and **E** is needed.

It is worth noting at this point that this schema requires a small change to the `retry` instruction in the original three stack WAM, i.e. when choice points and environments are allocated on the same stack. The usual code (on backtracking to a choice point \mathbf{B}) can set $\mathbf{EB} := \mathbf{B}$ while in CHAT this must become $\mathbf{EB} := \mathbf{B}[\mathbf{EB}]$.

As far as the complexity of this scheme of preserving environments is concerned, the same argument as in Section 3.1 for the heap applies. In the sequel we will refer to CHAT's technique of freezing a WAM stack without the use of freeze registers as *CHAT freeze*.

3.3 The Choice Point Stack and the Trail

CHAT borrows the mechanisms for dealing with the choice point stack and the trail from CAT: from the choice point stack, CAT copies only the consumer choice point. The reason is that at the moment that the consumer C is scheduled

to consume its answers, all the Prolog choice points (as well as possibly some generator choice points) will have exhausted their alternatives, and will have become redundant. This means that when a consumer choice point is reinstalled, this can happen immediately below a scheduling generator which is usually the leader of a scheduling component (see [2] for a more detailed justification why this is so). CHAT does exactly the same thing: it copies in what we call a *CHAT area* the consumer choice point. This copy is reinstalled whenever the consumer needs to consume more answers.

Also for the trail, CHAT is similar to CAT: the part of the trail between the consumer and the generator is copied, together with the values the trail entries point to. However, as also the heap and local stack are copied by CAT, CAT can make a selective copy of the trail, while CHAT must copy all of the trail between the consumer and the generator. This amounts to reconstructing the forward trail of the SLG-WAM (without back-pointers) for part of the computation.

For a single consumer, the cost of (partly) reconstructing the forward trail is not greater (in complexity) than what the SLG-WAM has incurred while maintaining this part of the forward trail. Figure 2 shows the state of CHAT immediately after encountering the consumer and doing all the actions described above; the shaded parts of the stacks show exactly the information that is copied by CHAT.

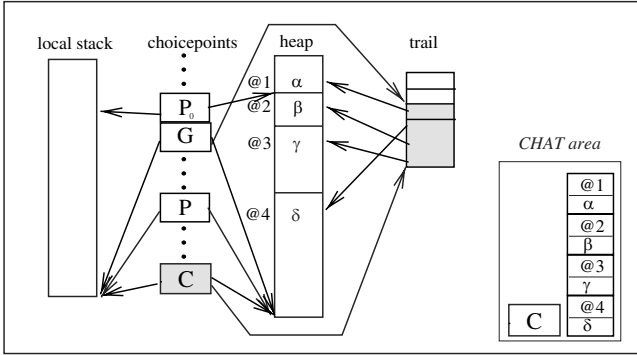


Fig. 2. Stacks and CHAT area after making a CHAT copy and adapting the choice points.

3.4 More Consumers and Change of Leader: A More Incremental CHAT

The situation with more consumers, as far as freezing the heap and local stack goes, is no different from that described above. Any time a new consumer is encountered, the **B**[EB] and **B**[H] fields of choice points **B** between the new consumer and its generator are adapted. Note that the same choice point can be adapted several times, and that the adapted fields can only point lower in the

corresponding stacks. From now on, we will drop the assumption that there is only one consumer.

It is also worth considering explicitly a coup: a change of leaders. Note that as far as the heap and local stack is concerned, nothing special needs to be done if each consumer performs CHAT freeze till its current leader at the time of its creation. For the trail, a similar mechanism as for CAT applies: an incremental part of the trail between the former and the new leader needs to be copied. In [2] it is shown that this need not be done immediately at the moment of the coup, but can be postponed until backtracking happens over a former leader so that the incremental copy can be easily shared between many consumers. It also leads directly to the same incremental copying principle as in CAT: each consumer needs only to copy trail up to the nearest generator and update this copy when backtracking over a non-leader generator occurs.

The incrementality of copying parts of the trail, also applies to the change of the EB and H fields in choice points: instead of adapting choice points up to the leader, one can do it up to the nearest generator. In this scheme, if backtracking happens over a non-leader generator, then its EB and H fields have to be propagated to all the choice points up to the next generator. Indeed, our implementation of CHAT employs both incremental copying of the trail and incremental adaptation of the choice points.

3.5 Reinstalling Consumers

As in CAT, CHAT can reinstall a single consumer C by copying the saved consumer choice point just below the choice point of a scheduling generator G . Let this copy happen at a point identified as \mathbf{B}_C in the choice point stack. The CHAT trail is reinstalled also exactly as in CAT by copying it from the CHAT area to the trail stack and reinstalling the bindings. There remains the installation of the correct top of heap and local stack registers: since the moment C was first saved in the CHAT area, it is possible that more consumers were frozen, and that these consumers are still suspended (i.e. their generators are not complete) when C is reinstalled. It means that C must protect also the heap of the other consumers. This is achieved by installing in \mathbf{B}_C the EB and H fields of G at the moment of reinstallation. This will lead to correctly protecting the heap, as G cannot be older than the leader of the still suspended consumers and G was in the active computation when the other consumers were frozen. Figure 3 gives a rough idea of the reinstallation of the execution environment of a single consumer; shaded parts of the stacks show the copied information.

The above describes a scheduling algorithm that schedules one consumer at a time. This contrasts with the current SLG-WAM scheduler which schedules in one pass all the consumers for which unconsumed answers are available. The CHAT scheduler can do the same as follows: the saved consumer choice points of all scheduled consumers C_1, \dots, C_k are copied one after the other in the choice point stack (again starting from immediately below the choice point of the generator G). The EB and H fields of these choice points reflect the corresponding

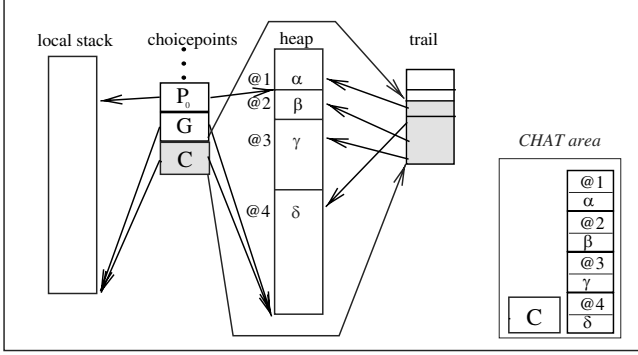


Fig. 3. Memory areas upon reinstalling the CHAT area for a single consumer C .

fields of G . However, only one of these consumers can have its execution environment (as represented by bindings in the corresponding CHAT trail area) reinstalled at any given point. Thus, only the last scheduled consumer C_k can immediately reinstall its trail and start consumption of its answers. The situation is depicted for $k = 3$ in Figure 4; the right part of the figure shows the ALT fields of choice points in detail. After all answers have been returned to C_k , this choice point is exhausted and execution fails back to the choice point of another scheduled consumer C_{k-1} ; at this point C_{k-1} through the use of a `chat_reinstall_trail` instruction reinstalls its trail *once* and starts consumption of answers through the `answer_return` instruction (see [4]). In short, upon failing to a consumer choice point whose $\mathbf{B}_C[\text{ALT}]$ is a `chat_reinstall_trail` instruction the action taken is as follows:

```

reinstall the trail from the CHAT trail area of the consumer in  $\mathbf{B}_C$ ;
save in  $\mathbf{B}_C[\text{TR}]$  the new top of trail stack;
alt :=  $\mathbf{B}_C[\text{ALT}]$  := answer_return;
goto alt;    /* transfer control to alt */

```

3.6 Releasing Frozen Space and the CHAT Areas upon Completion

The generator choice point of a leader is popped only at completion of its component. At that moment, the CHAT areas of the consumers that were led by this leader can be freed: this mechanism is again exactly the same as in CAT. Also, there are no more program clauses to execute for the completed leader and backtracking occurs to the previous choice point, say P_0 . P_0 contains the correct top of local stack and heap in its EB and H fields: these fields could have been updated in the past by CHAT or not. In either case they indicate the correct top of heap and local stack.

The SLG-WAM achieves this space reclamation at completion of a leader by resetting the freeze registers from the values saved in the leader choice point. Indeed, the SLG-WAM saves **HF**, **EF**, etc. in all generator choice points; see [4].

² If there is no previous choice point, the computation is finished.

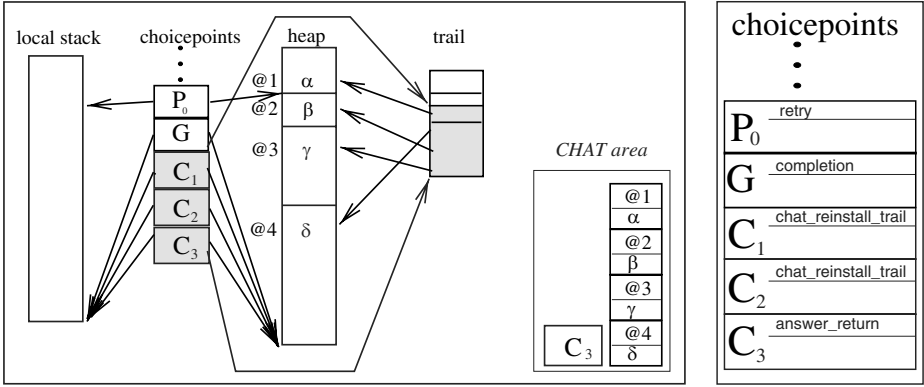


Fig. 4. Scheduling of three consumers C_1, C_2, C_3 and reinstallation of the trail of C_3 .

3.7 Handling of Negation in CHAT

Support for well-founded negation (implemented in the current version of CHAT) is an issue orthogonal to the mechanism used for suspension/resumption. Due to space limitations we do not describe how DELAYING is handled as it is similar to its handling by the SLG-WAM (see [5]), but restrict attention to fixed-order stratified negation as in [4]. The SLG-WAM, upon encountering a negative literal of an incomplete subgoal lays a negation suspension frame in the choice point stack and the current execution path is suspended by freezing the WAM stacks (see [4]). CHAT mimicks this behaviour by creating a CHAT area for the suspended computation, saving immediately the negation suspension frame there (i.e. without creating it on the CP stack) and incrementally saving its trail entries as in the case of a suspended consumer. Upon derivation of an answer for a subgoal with negation suspensions, the CHAT areas of these suspensions can be immediately reclaimed effectively failing the corresponding execution paths. The more interesting case is when a subgoal fails (i.e. gets completed with no answers): then all its negation suspension frames can be reinstalled through copying one after the other in the choice point stack, the topmost one can immediately reinstall its trail and continue with its saved forward continuation. The situation is completely analogous to the case of scheduling multiple consumers described in Section 3.5, notable differences are that 1) the negation suspension frames are not reinstalled below the choice point of the generator B_G , but, since for fixed-order stratified negation the subgoal is completed, starting from that point and 2) the possibility to immediately reclaim each CHAT area upon reinstallation of its trail, as this reinstallation only happens once. The second point is valid even in the case of non-stratified negation.

4 Best and Worst Cases

As noted in [2], a worst case for CAT can be constructed by making CAT copy and reinstall arbitrary often, arbitrary large amounts of heap to (and from) the CAT area. Since CHAT does not copy the heap, this same worst case does not apply. Still, CHAT can be made to behave arbitrarily worse than SLG-WAM. We also show an example in which the SLG-WAM uses arbitrary more space than CHAT.

4.1 The Worst Case for CHAT

There are two ways in which CHAT can be worse than SLG-WAM:

1. every time a consumer is saved, the choice point stack between the consumer and the leader is traversed; such an action is clearly not present in SLG-WAM neither CAT
2. trail chunks are copied by CHAT for each save of a consumer; the inefficiency lies in the fact that consumers in the SLG-WAM can share a part of the trail even strictly between the consumer and the nearest generator; this is a direct consequence of the forward trail with back pointers; both space and time complexity are affected. Note that the same source of inefficiency is present in CAT.

The following example program shows both effects. The subscripts g and c denote the occurrence of a subgoal that is a generator or consumer for $p(_)$.

```

query(Choices,Consumers) :-
    pg(_), make_choices(Choices,_),
    make_consumers(Consumers,[]).

make_choices(N,trail) :-
    N > 0, M is N - 1, make_choices(M,_).
make_choices(0,_).

make_consumers(N,Acc) :-
    N > 0, M is N - 1, pc(_), make_consumers(M,[a|Acc]).

:- table p/1.
p(1).

```

Predicate `make_choices/2` is supposed to create choice points; if the compiler is sophisticated enough to recognize that it is indeed deterministic, a more complicated predicate with the same functionality can be used. The reason for giving the extra argument to `make_consumers` is to make sure that on every creation of a consumer, **H** has a different value and an update of the **H** field of choice points between the new consumer and the generator is needed — otherwise, an obvious optimization of CHAT would be applicable. The query is e.g. `?- query(100,200)`. CHAT uses $(Choices * Consumers)$ times more space and time than SLG-WAM for this program. If the binding with the atom `trail` were not present in the above program, CHAT would also use $(Choices * Consumers)$ times more space and time than CAT.

At first sight, this seems to contradict the statement that CHAT is a better CAT. However, since for CHAT the added complexity is only related to the trail and choice points, the chances for running into this in reality are lower than for CAT.

4.2 A Best Case for CHAT

The best case space-wise for CHAT compared to SLG-WAM happens when lots of non tabled choice points get trapped under a consumer: in CHAT, they can be reclaimed, while in SLG-WAM they are frozen and retained till completion. The following program shows this:

```
query(Choices,Consumers) :-
    pg(_), create(Choices,Consumers), fail.
create(Choices,Consumers) :- Consumers > 0,
    ( make_choicepoints(Choices), pc(Y), Y = 2
      ; C is Consumers - 1, create(Choices,C) ).
make_choicepoints(C) :-
    C > 0, C1 is C - 1, make_choicepoints(C1).
make_choicepoints(0).
:- table p/1.
p(1).
```

When called with e.g. `?- query(25,77).` the maximal choice point usage of SLG-WAM contains at least $25 * 77$ Prolog choice points plus 77 consumer choice points; while CHAT's maximal choice point usage is 25 Prolog choice points (and 77 consumer choice points reside in the CHAT areas). Time-wise, the complexity of this program is the same for CHAT and SLG-WAM.

One should not exaggerate the impact of the best and worst cases of CHAT: in practice, such contrived programs rarely occur and probably can be rewritten so that the bad behaviour is avoided.

5 A Plethora of Implementations

After SLG-WAM and CAT, CHAT offers a third alternative for implementing the suspend/resume mechanism that tabled execution needs. It shares with CAT the characteristic that Prolog execution is not affected and with SLG-WAM the high sharing of execution environments of suspended computations. On the other hand, CHAT is not really a mixture of CAT and SLG-WAM: CHAT copies the trail in a different way from CAT and CHAT freezes the stacks differently from SLG-WAM, namely with the CHAT freeze technique. CHAT freeze can be achieved for the heap and local stack only. Getting rid of the freeze registers for the trail and choice point stacks can only be achieved by means of copying; the next section elaborates on this.

Thus, it seems there are three alternatives for the heap (SLG-WAM freeze, CHAT freeze and CAT copy) and likewise for the local stack, while there are

two alternatives for both choice point and trail stack (SLG-WAM freeze and CAT copy). The decisions on which mechanism to use for each of the four WAM stacks are independent. It means there are at least 36 possible implementations of the suspend/resume mechanism which is required for tabling !

It also means that one can achieve a CHAT implementation starting from the SLG-WAM as implemented in XSB, get rid of the freeze registers for the heap and the local stack, and then introduce copying of the consumer choice point and the trail. This was our first attempt: the crucial issue was that before making a complete implementation of CHAT, we wanted to have some empirical evidence that CHAT freeze for heap and local stack was correct. As soon as we were convinced of that, we implemented CHAT by partly recycling the CAT implementation of [2] which is also based on XSB as follows:

- replacing the selective trail copy of CAT with a full trail copy of the part between consumer and the closest generator
- not copying the heap and local stack to the CAT area while introducing the CHAT freeze for these stacks; this required a small piece of code that changes the H and EB entries in the affected choice points at CHAT area creation time and consumer reinstallation

It might have been nice to explore all 36 possibilities, with two or more scheduling strategies and different sets of benchmarks but unlike cats, we do not have nine lives.

6 More Insight

One might wonder why CHAT can achieve easily (i.e. without changing the WAM) the freezing of the heap and the local stack (just by changing two fields in some choice points) but the trail has to be copied and reconstructed. There are several ways to see why this is so. In WAM, the environments are already linked by back-pointers, while trail entries (or better trail entry chunks) are not. Note that SLG-WAM does link its trail entries by back-pointers; see [4]. Another aspect of this issue is also typical to an implementation which uses untrailing (instead of copying) for backtracking (or more precisely for restoring the state of the abstract machine): it is essential that trail entry chunks are delimited by choice points; this is not at all necessary for heap segments. Finally, one can also say that CHAT avoids the freeze registers by installing their value in the affected choice points: The WAM will continue to work correctly, if the H fields in some choice points are made to point lower in the heap. The effect is just less reclamation of heap upon backtracking. Similarly for the local stack. On the other hand, the TR fields in choice points cannot be changed without corrupting backtracking.

7 Tests

All measurements were conducted on an Ultra Sparc 2 (168 MHz) under Solaris 2.5.1. Times are reported in seconds, space in KBytes. Space numbers mea-

sure the maximum use of the stacks (for SLG-WAM) and the total of max. stack + max. C(H)AT area (for C(H)AT) ³. The size of the table space is not shown as it is independent of the suspension/resumption mechanism that is used. The benchmark set is exactly the same as in [2] where more information about the characteristics of the benchmarks and the impact of the scheduling can be found. We also note that the versions of CAT and CHAT we used did not implement the multiple scheduling of consumers described in Section 3.5, while the SLG-WAM scheduling algorithms scheduled all consumers in one pass and was thus invoked less frequently.

7.1 A Benchmark Set Dominated by Tabled Execution

Programs in this first benchmark set perform monomorphic type analysis by tabled execution of type-abstracted input programs. Minimal Prolog execution is going on as tabling is also used in the domain-dependent abstract operations to avoid repeated subcomputations. Tables 1 and 2 show the time and space performance of SLG-WAM, CHAT and CAT for the batched (indicated by B in the tables) and local scheduling strategy (indicated by L).

	cs_o	cs_r	disj_o	gabriel	kalah_o	peep	pg	read_o
SLG-WAM(B)	0.23	0.45	0.13	0.17	0.15	0.44	0.12	0.58
CHAT(B)	0.21	0.42	0.13	0.15	0.15	0.46	0.14	0.73
CAT(B)	0.22	0.41	0.13	0.15	0.14	0.50	0.15	0.92
SLG-WAM(L)	0.23	0.43	0.13	0.16	0.16	0.42	0.12	0.61
CHAT(L)	0.22	0.42	0.12	0.15	0.14	0.40	0.11	0.53
CAT(L)	0.22	0.42	0.12	0.15	0.14	0.40	0.11	0.55

Table 1. Time performance of SLG-WAM, CAT & CHAT under batched & local scheduling.

For the local scheduling strategy, CAT and CHAT perform the same time-wise and systematically better than SLG-WAM. Under the batched scheduling strategy, the situation is less clear, but CHAT is never worse than the other two. Taking into account the uncertainty of the timings, it is fair to say that except for `read_o` all three implementation schemes perform more or less the same time-wise in this benchmark set for batched scheduling.

As can be seen in Table 2, the local scheduling strategy has a clear advantage in space consumption in this benchmark set: the reason is that its scheduling components are tighter than those of batched scheduling — we refer to [2] for additional measurements on why this is so. Space-wise, CHAT wins always from

³ It is worth noting here that since conducting the performance tests we have found places in the XSB code where memory reclamation could be improved. Although this affects all implementation schemes, it probably does not affect them equally, so the space figures should be taken *cum grano salis*.

	cs_o	cs_r	disj_o	gabriel	kalah_o	peep	pg	read_o
SLG-WAM(B)	9.7	11.4	8.8	20.6	40	317	119	512
CHAT(B)	9.6	11.6	8.4	24.7	35.1	770	276	1080
CAT(B)	13.6	19.4	11.7	45.3	84	3836	1531	5225
SLG-WAM(L)	6.7	7.6	5.8	17.2	13.3	19	15.8	93
CHAT(L)	5.8	7.2	5.6	19	8.2	16	13.2	101
CAT(L)	7.9	10.7	7.1	29.5	12.5	17	23.5	246

Table 2. Space performance of SLG-WAM, CAT & CHAT under batched & local scheduling.

CAT and 6 out of 8 times from SLG-WAM (using local scheduling). Indeed, most often the extra space that CHAT uses to copy trail entry chunks is compensated for by the lower choice point stack consumption.

7.2 A More Realistic Mix of Tabled and Prolog Execution

Programs in this second benchmark set are also from an application of tabling to program analysis: a different abstract domain is now used and although tabling is necessary for the termination of the fixpoint computation, the domain-dependent abstract operations do not benefit from tabling as they do not contain repeated (i.e. identical) subcomputations; they are thus implemented in plain Prolog. As a result, in this set of benchmarks 75–80% of the execution concerns Prolog code. We consider this mix a more “typical” use of tabling. We note at this point that CHAT (and CAT) have faster Prolog execution than SLG-WAM by around 10% according to the measurements of [4] — this is the overhead that the SLG-WAM incurs on the WAM. In the following tables all figures are for the local scheduling strategy; batched scheduling does not make sense for this set of benchmarks as the analyses are based on an abstract least upper bound (lub) operation. For lub-based analyses, local scheduling bounds the propagation of intermediate (i.e. not equal to the lub) results to considerably smaller components than those of batched: in this way, a lot of unnecessary computation is avoided.

	akl	color	bid	deriv	read	browse	serial	rdtok	boyer	plan	peep
SLG-WAM	1.48	0.67	1.11	2.56	9.64	32.6	1.17	3.07	10.02	7.61	9.01
CHAT	1.25	0.62	1.03	2.54	9.73	32	0.84	2.76	10.17	6.14	8.65
CAT	1.24	0.62	0.97	2.50	9.56	32.2	0.83	2.75	9.96	6.38	8.54

Table 3. Time Performance of SLG-WAM, CHAT & CAT.

Table 3 shows that CAT wins on average over the other two. CHAT comes a close second: in fact CHAT’s performance in time is usually closer to those of CAT than those of SLG-WAM. Space-wise — see Table 4 — CHAT wins from both SLG-WAM and CAT in all benchmarks. It has lower trail and choice point stack consumption than SLG-WAM and as it avoids copying information from

	akl	color	bid	deriv	read	browse	serial	rdtok	boyer	plan	peep
SLG-WAM	998	516	530	472	5186	9517	279	1131	2050	1456	1784
CHAT	433	204	198	311	4119	7806	213	746	819	963	1187
CAT	552	223	206	486	8302	7847	227	821	1409	1168	1373

Table 4. Space Performance (in KBytes) of SLG-WAM, CHAT & CAT.

the local stack and the heap, it saves considerably less information than CAT in its copy area.

8 Conclusion

CHAT offers one more alternative to the implementation of the suspend/resume mechanism that tabling requires. Its main advantage over SLG-WAM's approach is that no freeze registers are needed and in fact no complicated changes to the WAM. As with CAT, the adoption of CHAT as a way to introduce tabling to an existing logic programming system does not affect the underlying abstract machine and the programmer can still rely on the full speed of the system for non-tabled parts of the computation. Its main advantage over CAT is that CHAT's memory consumption is lower and much more controlled. The empirical results show that CHAT behaves quite well and also that CHAT is a better candidate for replacing SLG-WAM (as far as the suspension/resumption mechanism goes) than CAT. CHAT also offers the same advantages as CAT as far as flexible scheduling strategies goes.

Acknowledgements

The research of the second author is supported by a junior scientist fellowship from the K.U. Leuven Research Council and CCR-9702681.

References

- [1] H. Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press, Cambridge, Massachusetts, 1991. See also: <http://www.isg.sfu.ca/~hak/documents/wam.html>.
- [2] B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Principles of Declarative Programming, 10th International Symposium, PLILP'98, Held Jointly with the 6th International Conference, ALP'98*, number 1490 in LNCS, pages 21–35, Pisa, Italy, Sept. 1998. Springer.
- [3] B. Demoen and K. Sagonas. Memory Management for Prolog with Tabling. In *Proceedings of ISMM'98: ACM SIGPLAN International Symposium on Memory Management*, pages 97–106, Vancouver, B.C., Canada, Oct. 1998. ACM Press.
- [4] K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *ACM Transactions on Programming Languages and Systems*, 20(3), May 1998.

- [5] K. Sagonas, T. Swift, and D. S. Warren. An Abstract Machine for Computing the Well-Founded Semantics. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 274–288, Bonn, Germany, Sept. 1996. The MIT Press.
- [6] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, Menlo Park, U.S.A., Oct. 1983.

The Influence of Architectural Parameters on the Performance of Parallel Logic Programming Systems

Marcio G. Silva¹, Inês C. Dutra¹, Ricardo Bianchini¹, and Vítor Santos Costa²

¹ COPPE/Systems Engineering
Federal University of Rio de Janeiro, Brazil
{mgs, ines, ricardo}@cos.ufrj.br

² LIACC and DCC-FCUP
4150 Porto, Portugal
vsc@ncc.up.pt

Abstract. In this work we investigate how different machine settings for a hardware Distributed Shared Memory (DSM) architecture affect the performance of parallel logic programming (PLP) systems. We use execution-driven simulation of a DASH-like multiprocessor to study the impact of the cache block size, the cache size, the network bandwidth, the write buffer size, and the coherence protocol on the performance of Andorra-I, a PLP system capable of exploiting implicit parallelism in Prolog programs. Among several other observations, we find that PLP systems favour small cache blocks regardless of the coherence protocol, while they favour large cache sizes only in the case of invalidate-based coherence. We conclude that the cache block size, the cache size, the network bandwidth, and the coherence protocol have a significant impact on the performance, while the size of the write buffer is somewhat irrelevant.

Keywords: DSM architectures, performance evaluation, logic programming

1 Introduction

Parallel logic programming (PLP) systems are sophisticated examples of symbolic computing systems. They address problems such as dynamic memory allocation, scheduling irregular execution patterns, and managing different types of implicit parallelism. Most PLP systems have been developed for bus-based shared-memory architectures. Their complexity and the large amount of data they process raise the question of whether PLP systems can still obtain good performance on scalable architectures, such as hardware distributed shared-memory (DSM) systems.

Our previous work [16, 15, 17] has shown that PLP systems as originally implemented do not perform well on scalable DSM multiprocessors; PLP systems only perform well for these machines when heavily modified to improve locality

of reference. We focused on Andorra-I [18], a PLP system capable of exploiting implicit parallelism in Prolog programs, that was originally developed for the Sequent Symmetry bus-based multiprocessor. Our work has shown that Andorra-I requires several data structure modifications to run well on a DASH-like multiprocessor. However, our work has not addressed how different architectural parameters affect the performance of PLP systems on scalable multiprocessors. Thus, in this work we investigate the impact of the cache block size, the cache size, the network bandwidth, the processor’s write buffer size, and the coherence protocol on the performance of the modified version of Andorra-I, again running on a simulated DASH-like multiprocessor.

Among several other observations, we find that Andorra-I favours small cache blocks regardless of the coherence protocol, while it favours large cache sizes only in the case of invalidate-based coherence. In addition, Andorra-I can profit from high network bandwidth, especially when update-based coherence is used. Based on our results, we conclude that Andorra-I can benefit from large cache sizes, small to medium cache blocks, and high network bandwidth. The depth of the write buffer does not seem to affect performance significantly.

Our work contrasts with similar studies by other researchers. Montelius [13] and Tick and Hermenegildo [19], for instance, studied the performance of PLP systems implemented for bus-based multiprocessors, while we are more interested on how these implementations run on a scalable multiprocessor and the modifications they require in order to perform well. Other researchers have studied the performance of PLP systems on scalable architectures, such as the DDM [14], but did not evaluate the impact of different machine settings and different coherence protocols.

The remainder of this paper is organised as follows. Section 2 describes our methodology and benchmarks. Section 3 discusses our results. Section 4 draws our conclusions and directions for future work.

2 Methodology

2.1 Andorra-I

Andorra-I is a parallel logic programming system that exploits determinate dependent and-parallelism and or-parallelism arising from non-determinate goals. Its implementation is influenced by JAM [3] when exploiting and-parallelism, and by Aurora [11] when exploiting or-parallelism.

In Andorra-I, a processing element that performs computation is called a *worker*. In practice, each worker corresponds to a separate processor. Andorra-I is designed in such a way that workers are classified as *masters* or *slaves*. One master and zero or more slaves form a *team*. Each master in a team is responsible for creating a new choicepoint, while slaves are managed and synchronised by their master. Workers in a team cooperate with each other in order to share available and-work. Different teams of workers cooperate to share or-work.

Workers should perform reductions most of the execution time, i.e., they should spend most of their time executing *engine* code. Andorra-I is designed

so that data corresponding to each worker is as local as possible, so that each worker tries to find its own work without interfering with others. Scheduling in Andorra-I is demand-driven, that is, whenever a worker runs out of work, it enters a *scheduler* to find another piece of available work.

The or-scheduler is responsible for finding or-work, i.e. an unexplored alternative in the or-tree. Our experiments used the Bristol or-scheduler [1], originally developed for Aurora. The and-scheduler is responsible for finding eligible and-work, which corresponds to a goal in the run queue (list of goals not yet executed) of a worker in the same team. Each worker in a team keeps a run queue of goals. This run queue of goals has two pointers. The pointer to the head of the queue is only used by the owner. The pointer to the tail of the queue is used by other workers to “steal” goals when their own run queues are empty. If all the run queues are empty, the slaves wait either until some other worker (in our implementation, the master) creates more work in its run queue or until the master detects that there are no more determinate goals to be reduced and it is time to create a choicepoint. The reconfigurer is responsible for moving slaves and masters between teams when they do not find any more work on their own teams [5].

The version of Andorra-I used for our experiments is an optimised version that employs some data structure-level techniques (such as padding, alignment, and lock distribution) to perform well on scalable DSM machines [17].

2.2 Multiprocessor Simulation

We use a detailed on-line, execution-driven simulator that simulates a 32-node, DASH-like [10], directly-connected multiprocessor. Each node of the simulated machine contains a single processor, a write buffer, a direct-mapped data cache, local memory, a full-map directory, and a network interface. The simulator was developed at the University of Rochester and uses the MINT front-end, developed by Veenstra and Fowler [20], to simulate the MIPS architecture, and a back-end, developed by Bianchini and Veenstra, to simulate the memory and interconnection systems.

In our simulated machine, all instructions and read hits are assumed to take 1 processor cycle. Read misses stall the processor until the read request is satisfied. Writes go into a write buffer and take 1 cycle, unless the write buffer is full, in which case the processor stalls until an entry becomes free. Reads are allowed to bypass writes that are queued in the write buffers. Shared data are interleaved across the memories at the block level.

A memory bus clocked at half of the speed of the processor connects the main components of each machine node. A new bus operation can start every 34 processor cycles. A memory module can provide the first word of a cache line 20 processor cycles after the request is issued. The other words are delivered at 2 cycles/word bandwidth.

The interconnection network is a bi-directional wormhole-routed mesh, with dimension-ordered routing. The network clock speed is the same as the processor

clock speed. Switch nodes introduce a 4-cycle delay to the header of each message. In these networks contention for links and buffers is captured at the source and destination of messages. All our architectural parameters are compatible with those of scalable multiprocessors such as DASH.

In order to keep caches coherent we used write-invalidate (WI) [7] and write-update (WU) [12] protocols. In the WI protocol, whenever a processor writes a data item, copies of the cache block containing the item in other processors' caches are invalidated. If one of the invalidated processors later requires the same item, it will have to fetch it from the writer's cache. Our WI protocol keeps caches coherent using the DASH protocol with release consistency [9].

WU protocols are the main alternative to invalidate-based protocols. In WU protocols, whenever an item is written, the writer sends copies of the new value to the other processors that share the item. In our WU implementation, a processor writes through its cache to the home node. The home node sends updates to the other processors sharing the cache block, and a message to the writing processor containing the number of acknowledgements to expect. Sharing processors update their caches and send an acknowledgement to the writing processor. The writing processor only stalls waiting for acknowledgements at a lock release point.

2.3 Applications

The benchmarks we used in this work are applications representing predominantly and-parallelism, predominantly or-parallelism, and both and- and or-parallelism.

All results, except for the and/or-parallel application, were obtained using a static configuration of workers. The or-parallel applications used a fixed all-masters configuration and the and-parallel applications used a fixed one-team of one master and slaves. Our results correspond to the *first* run of an application (results would be somewhat better for other runs).

And-Parallel Application: As an example and-parallel application, we used the clustering algorithm for network management from British Telecom. The program receives a set of points in a three dimensional space and groups these points into *clusters*. Basically, three points belong to the same cluster if the distance between them is smaller than a certain limit. And-parallelism in this application naturally stems from running the calculations for each point in parallel. The test program uses a cluster of 400 points as input data.

Or-Parallel Applications: We used two programs as or-parallel applications. The first is an example from the well-known natural language question-answering system **chat-80**, written at the University of Edinburgh by Pereira and Warren. This version of **chat-80** operates on the domain of world geography. The program **chat** makes queries to the **chat-80** database. This is a small scale benchmark with good or-parallelism, and it has been traditionally used as one of the or-parallel benchmarks for both the Aurora and Muse systems.

The second program solves the popular N-queens problem. Given a chessboard $N \times N$, the problem consists of placing all queens on the chessboard in a

way that no queen can attack each other in a row, column or diagonal. This program has a high degree of or-parallelism and is also used as benchmark for both the Aurora and Muse systems.

And/Or-Parallel Application: We used a program to generate naval flight allocations, based on a system developed by Software Sciences and the University of Leeds for the Royal Navy. It is an example of a real-life resource allocation problem. The program allocates airborne resources (such as aircraft) whilst taking into account a number of constraints. The problem is solved by using the technique of active constraints as first implemented for Pandora. The program has both or-parallelism and and-parallelism. The input data we used for testing the program consists of 11 aircrafts, 36 crew members, and 10 flights to be scheduled. The degree of and- and or-parallelism in this program varies according to the queries, but the query used produces more and-parallelism than or-parallelism.

More details and references for the applications can be found elsewhere [6].

2.4 Read Misses Categorisation

For each application, performance is directly related to cache read miss rates and network traffic caused by each change in the machine setting and in the coherence protocol.

For the WI protocol, the simulator classifies misses as:

- **Cold start misses.** A cold start miss happens on the first reference to a block by a processor.
- **True sharing misses.** A true sharing miss happens when a processor references a word belonging in a block it had previously cached but has been invalidated, due to a write by some other processor to the same word.
- **False sharing misses.** A false sharing miss occurs in roughly the same circumstances as a true sharing miss, except that the word written by the other processor is not the same as the word missed on.
- **Eviction misses.** An eviction (replacement) miss happens when a processor replaces one of its cache blocks with another one mapping to the same cache line and later needs to reload the block replaced.

This classification uses the algorithm described in [4], as extended in [2] by Bianchini and Kontothanassis. Note that the WU protocol does not involve sharing misses.

We classify updates as either *useful* (also known as **true sharing updates**), which are needed for correct execution of the program, or *useless*. The former category is used when the receiving processor references the word modified by the update message before another update message to the same word is received. The latter category of updates includes [2]:

- **False sharing updates.** The receiving processor does not reference the word modified by the update message before it is overwritten by a subsequent update, but references some other word in the same cache block.

- **Proliferation updates.** The receiving processor does not reference any of the words in the cache block until the next update for the same word.
- **Replacement updates.** The receiving processor does not reference the updated word until the block is replaced in its cache.
- **Termination updates.** A termination update is a proliferation update happening at the end of the program.

3 Results and Analysis

We simulate Andorra-I for all applications assuming 16 processors. For each application we vary cache sizes, cache block sizes, number of write buffer entries, network bandwidth, and coherence protocol. We vary each of these parameters at a time, keeping our default values for the other parameters (64-byte blocks, 64-KByte caches, network link bandwidth of 200 MBytes/sec, and 16-entry write buffers).

3.1 Varying Cache Block Sizes

Figure 1 shows execution times (in processor cycles) for our applications using WI and WU protocols. First of all, we can observe from the figure that the WI protocol is clearly the best for the applications with and-parallelism, while the influence of the protocol is not as significant for the or-parallel applications up to 128-byte blocks.

We can also observe that in general, small cache blocks (16, 32) yield better performance than larger cache blocks. In the best case, small block sizes can speedup the application as significantly as 5-fold with respect to 256-byte blocks. This is the case of the `chat-80` application under the WU protocol, which achieves its fastest execution with 64-byte blocks (1.75 million cycles) and slowest execution with 256-byte blocks (9.44 million cycles).

Part of the explanation for these results comes from the miss rates entailed by these block sizes and protocols. Figure 2 categorizes cache misses as a function of the block size and coherence protocol. The figure shows that the number of cold misses decreases as we increase the block size, as one would expect, since larger blocks transfer more data per miss into the caches. However, cold misses is the only category that consistently decreases with increases in block size. 256-byte blocks, for instance, increase the miss rate of all applications (with respect to 128-byte blocks) under the WI protocol, while they only increase the miss rate of the applications with and-parallelism under WU.

The main problem here is that larger blocks often increase the number of cache evictions and cause significant false sharing. Note however that false sharing manifests itself differently, depending on the coherence protocol. Under WI, false sharing increases the number of coherence (false) misses, while under WU false sharing simply increases the amount of coherence traffic (false and proliferation updates). These increases can be seen in figures 3 and 4.

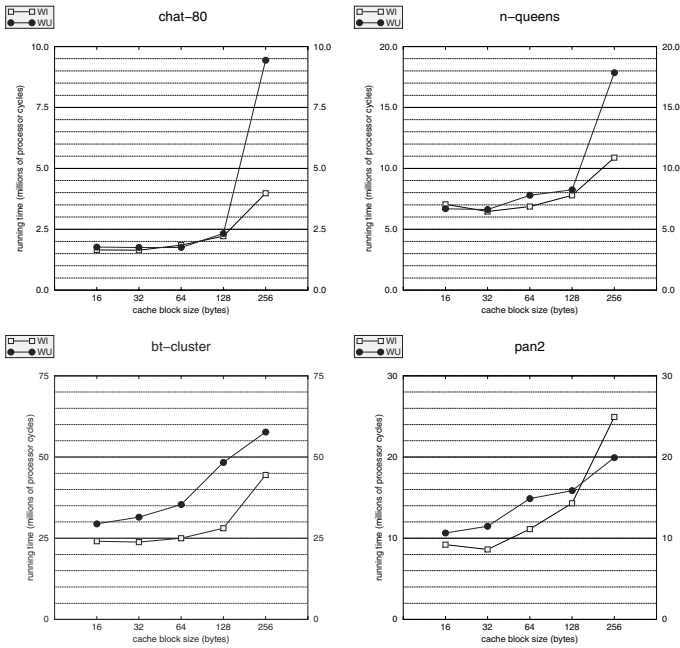


Fig. 1. VARYING CACHE BLOCK SIZE, EXECUTION TIMES

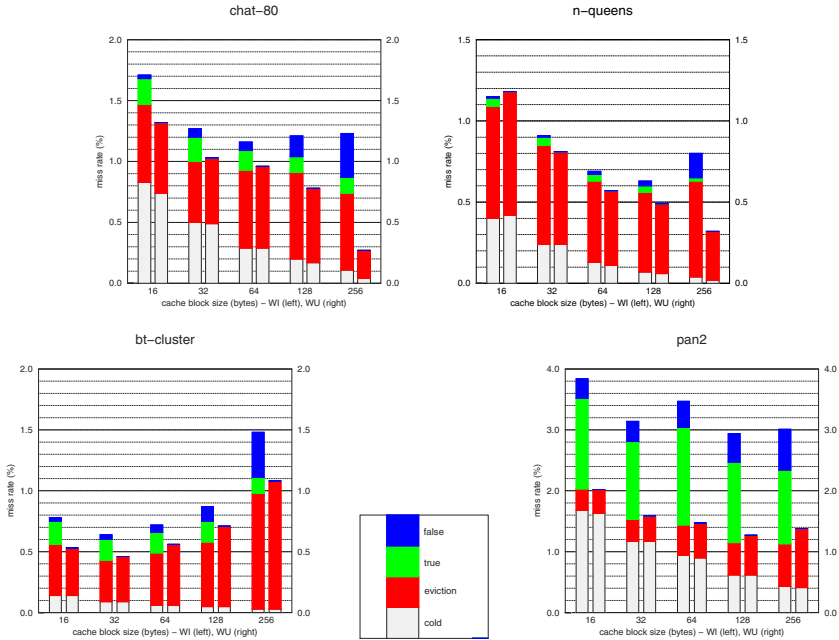


Fig. 2. VARYING CACHE BLOCK SIZE, MISS RATE

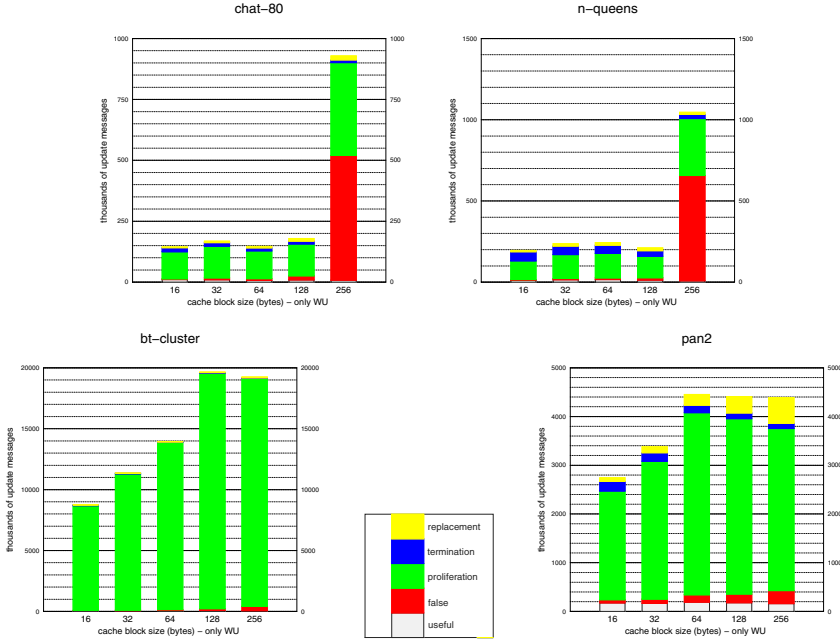


Fig. 3. VARYING CACHE BLOCK SIZE, CATEGORISING UPDATE MSGS

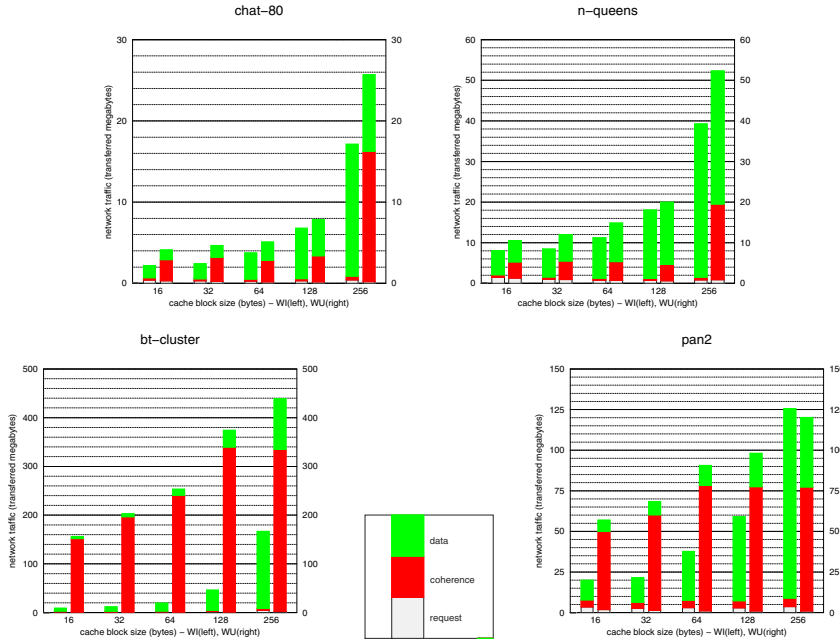


Fig. 4. VARYING CACHE BLOCK SIZE, NETWORK TRAFFIC

Figure 3 categorizes the update messages involved in each application as a function of block size. The figure explains the poor behaviour of the or-parallel applications under 256-byte blocks and the WU protocol. False sharing seriously increases the amount of coherence traffic in these applications, particularly for 256-byte blocks. This false sharing comes from the or-scheduler data structures used to manage the choicepoint stack. The applications with and-parallelism communicate much more intensively than the or-parallel applications under WU, but do not suffer as severely from false sharing.

The growth in the number of eviction and false misses and useless update messages generates the network traffic displayed in figure 4. Under the WI protocol, the larger the cache block, the larger the data messages being exchanged to service cache misses. Under WU, the network traffic grows significantly when compared to the WI traffic. As the block size increases, the amount of traffic, mainly coherence traffic, also grows. For the two or-parallel applications the traffic grows so intense that the network saturates. An experiment with chat-80 using 256-byte blocks and assuming infinite network bandwidth confirms this by decreasing the execution time from about 9 million cycles (shown in figure 11) to 3.7 million cycles.

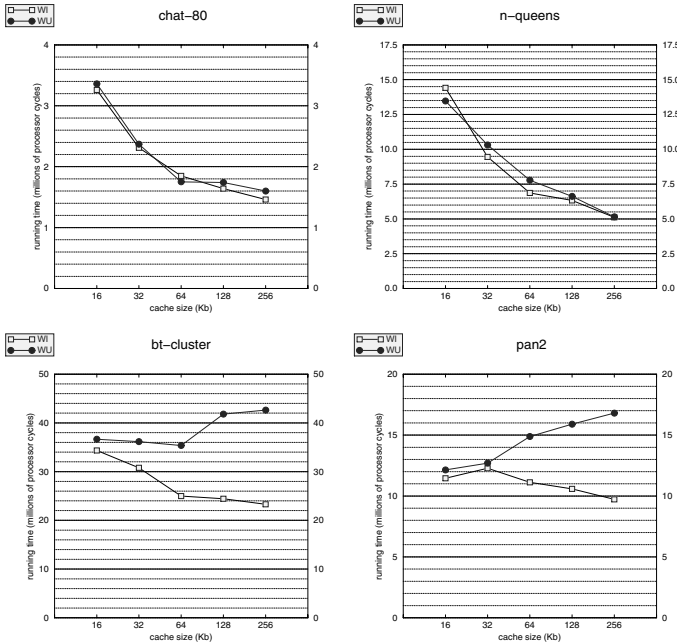


Fig. 5. VARYING CACHE SIZES, EXECUTION TIMES

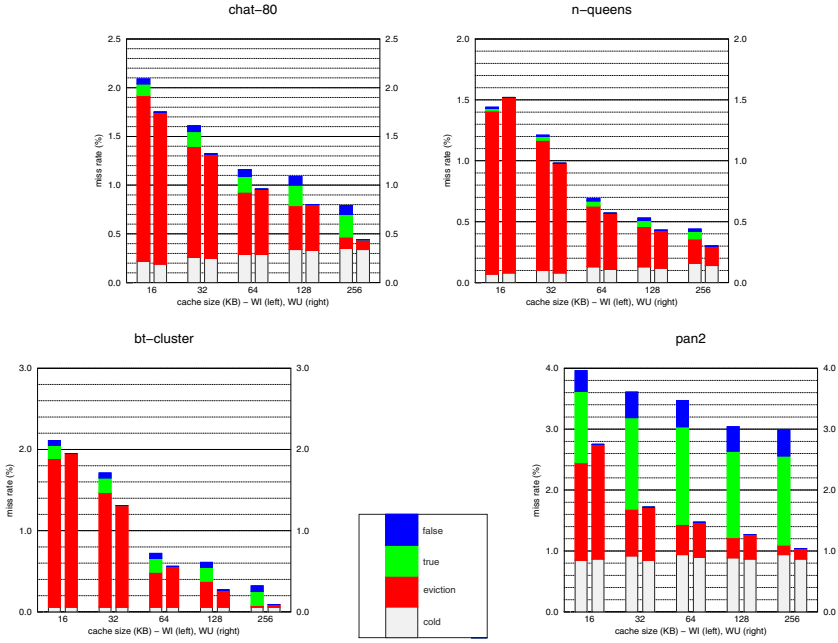


Fig. 6. VARYING CACHE SIZES, MISS RATES

3.2 Varying Cache Sizes

Figure 5 shows execution times in number of processor cycles for our applications as a function of different cache sizes. The figure shows that variations in cache size from 16 to 256 KBytes can speedup an application by as much as 2.8 times. This is the case of the *N-queens* application under the WI protocol, which executes in 14.41 million cycles as its worst time and 5.11 million cycles as its best time.

The figure also shows that the or-parallel applications achieve their best performance with 256 KBytes of cache, regardless of the coherence protocol. In contrast, applications with predominantly and-parallelism are severely affected by the kind of protocol used. For the WI protocol, the best cache size is still 256 KBytes, but for WU, smaller caches produce better performance. The reason for this latter result is that larger caches increase the degree of sharing, since fewer shared cache blocks are evicted. This higher degree of sharing means an increased number of update messages, which in turn causes an increase in the amount of network traffic involved in our applications. The applications with and-parallelism suffer more significantly from the larger caches, since they involve a larger amount of sharing between processes and communicate much more intensively than the or-parallel applications.

As one would expect, figure 6 shows that the read miss rates decrease with the increase in cache size, as a result of sharp decreases in the number of eviction misses. However, under the WU protocol, these decreases come at the cost of

a substantial increase in proliferation updates, particularly for the applications with and-parallelism, as can be observed in figures 7 and 8.

This result is mostly a consequence of how processes share data in Andorra-I. More specifically, or-parallel applications have an access pattern of the type single-writer/multiple-readers for the local, heap, and trail stacks, and multiple-writers/multiple-readers for the choicepoint stack, which is much more rarely written than the other stacks. In contrast, in and-parallel applications processors read and write to the shared stacks (local, heap, and trail) and to the run queue of parallel goals.

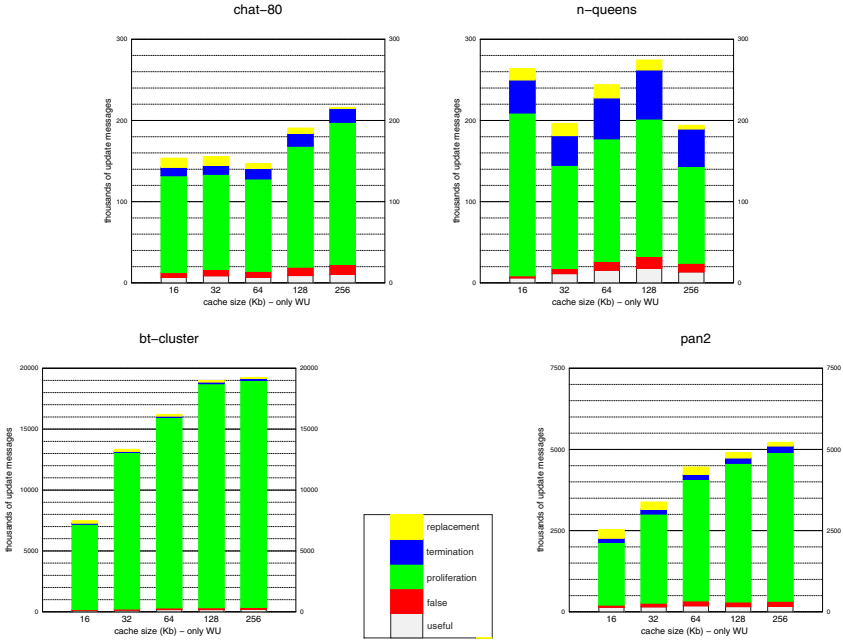


Fig. 7. VARYING CACHE SIZES, CATEGORISING UPDATE MSGS

3.3 Varying Network Bandwidth

Figure 9 shows the performance of Andorra-I for our applications when varying the network bandwidth through variations in the network path width (our default path width, 16 bits, corresponds to a 200 MBytes/sec link bandwidth). As can be observed in the figure, and was expected, the larger the path width, the better the execution time, independently of the coherence protocol. An increase in network path width from 8 to 32 bits can make an application run 1.95 times faster. This is the case of the pan2 application under the WU protocol, where the worst running time is 19.50 million cycles and the best time is 10.19 million cycles.

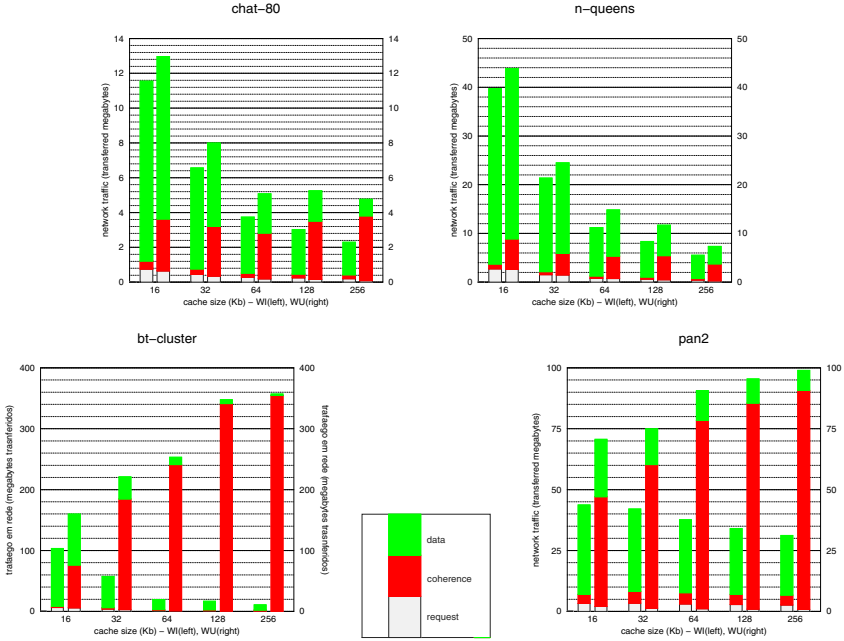


Fig. 8. VARYING CACHE SIZES, NETWORK TRAFFIC

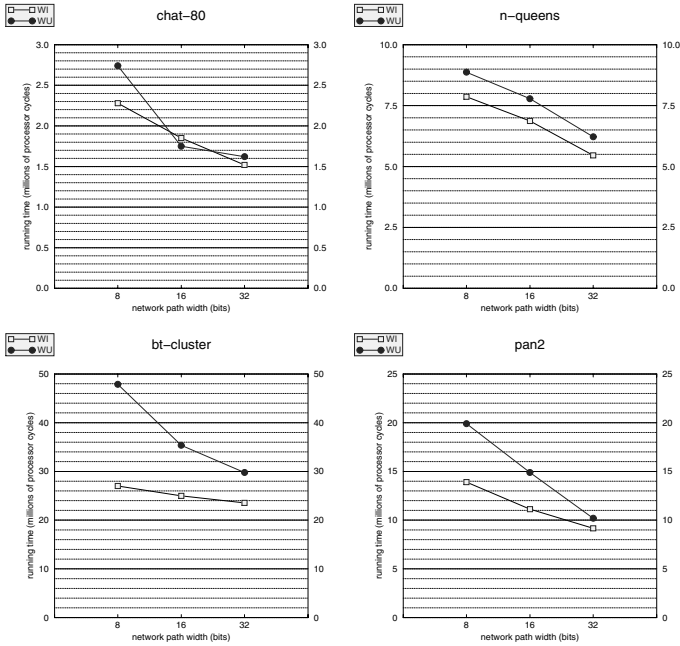


Fig. 9. VARYING NET PATH WIDTH, EXECUTION TIMES

The performance improvements produced by the wider paths are more significant for the WU protocol, as it involves a more intense communication traffic than WI. However, not even our widest network paths were enough for WU to significantly outperform WI, regardless of the type of application.

3.4 Varying Write Buffer Sizes

We also varied the size of write buffers from 4 to 64 entries. This variation impacts the system performance only negligibly. Even when the percentage of writes is relatively high, the variations in write buffer size never produce more than a 5% difference in performance.

3.5 Summary Results

Our results show the architectural parameters have a significant impact on the performance of PLP systems such as Andorra-I. For example, Andorra-I is very efficient under the WI protocol, large cache sizes (128 or 256 KBytes), small cache blocks (16 or 32 bytes), and wide network paths (32 bits), but suffers somewhat when these parameters are not observed. The WU protocol is competitive with the WI protocol for the or-parallel applications, while it produces poor performance for the and-parallel applications. This is mainly due to the way and-parallelism is exploited in systems such as Andorra-I, where workers share most of the data structures and generate significant coherence traffic.

4 Conclusions

This work showed that parallel logic programming systems originally written for bus-based shared-memory machines can run with good performance on a scalable multiprocessor, provided that the architectural parameters of the multiprocessor are appropriate. More specifically, we simulated the Andorra-I system on a DASH-like scalable multiprocessor with different machine settings and coherence protocols. We used four different applications with diverse types of parallelism. We have shown that the performance of Andorra-I can be greatly affected by different architectural parameters. For instance, under the write-invalidate protocol, Andorra-I can benefit from large cache sizes, small cache blocks, and wide network paths. The depth of the write buffer does not affect performance significantly.

In general, invalidate-based protocols favour and-parallel applications while or-parallel applications are not much affected by either protocol, with some exceptions. We conclude then that a combination of invalidate and update protocols and a combination of the best parameters studied in this work could yield best performance.

We believe that our results will hold for any parallel logic programming system designed for shared-memory machines. We intend to repeat these experiments for Aurora [11], a logic programming system that only exploits or-parallelism. We also intend to study the behaviour of more applications in detail

by investigating the impact of alternative cache coherence protocols on each shared data area in the system. Next we will tackle the problem of getting maximum performance of parallel logic programming systems on modern architectures by changing the structural design of the algorithms.

Acknowledgments

The authors would like to thank the CNPq Brazilian Research Council for its support. Vítor Santos Costa is thankful to PRAXIS and Fundação para a Ciência e Tecnologia for their financial support. This work was partially funded by the Melodia project (JNICT PBIC/C/TIT/2495/95).

References

- [1] Anthony Beaumont, S. Muthu Raman, and Péter Szeredi. Flexible Scheduling of Or-Parallelism in Aurora: The Bristol Scheduler. In Aarts, E. H. L. and van Leeuwen, J. and Rem, M., editor, *PARLE91: Conference on Parallel Architectures and Languages Europe*, volume 2, pages 403–420. Springer Verlag, June 1991. Lecture Notes in Computer Science 506.
- [2] R. Bianchini and L. I. Kontothanassis. Algorithms for categorizing multiprocessor communication under invalidate and update-based coherence protocols. In *Proceedings of the 28th Annual Simulation Symposium*, April 1995.
- [3] J. A. Crammond. The Abstract Machine and Implementation of Parallel Parlog. Technical report, Dept. of Computing, Imperial College, London, June 1990.
- [4] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenstrom. The detection and elimination of useless misses in multiprocessors. In *Proceedings of the 20th ISCA*, pages 88–97, May 1993.
- [5] I. C. Dutra. Strategies for Scheduling And- and Or-Work in Parallel Logic Programming Systems. In *Proceedings of the 1994 International Logic Programming Symposium*, pages 289–304. MIT Press, 1994. Also available as technical report CSTR-94-09, from the Department of Computer Science, University of Bristol, England.
- [6] I. C. Dutra. *Distributing And- and Or-Work in the Andorra-I Parallel Logic Programming System*. PhD thesis, University of Bristol, Department of Computer Science, February 1995. available at <http://www.cos.ufrj.br/~ines>.
- [7] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th International Symposium on Computer Architecture*, pages 124–131, 1983.
- [8] Markus Hitz and Erich Kaltofen, editors. *Proceedings of the Second International Symposium on Parallel Symbolic Computation, PASCO'97*, July 1997.
- [9] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. *Proceedings of the 17th ISCA*, pages 148–159, May 1990.
- [10] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The dash prototype: Logic overhead and performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, Jan 1993.
- [11] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora Or-parallel Prolog System. *New Generation Computing*, 7(2,3):243–271, 1990.

- [12] E. M. McCreight. The Dragon Computer System, an Early Overview. In *NATO Advanced Study Institute on Microarchitecture of VLSI Computers*, July 1984.
- [13] Johan Montelius and Seif Haridi. An evaluation of Penny: a system for fine grain implicit parallelism. In *Proceedings of 2nd International Symposium on Parallel Symbolic Computation* [8], July 1997.
- [14] S. Raina, D. H. D. Warren, and J. Cownie. Parallel Prolog on a Scalable Multiprocessor. In Peter Kacsuk and Michael J. Wise, editors, *Implementations of Distributed Prolog*, pages 27–44. Wiley, 1992.
- [15] V. Santos Costa, Bianchini, and I. C. Dutra. Parallel Logic Programming Systems on Scalable Multiprocessors. In *Proceedings of the 2nd International Symposium on Parallel Symbolic Computation, PASCO'97* [8], pages 58–67, July 1997.
- [16] V. Santos Costa, R. Bianchini, and I. C. Dutra. Evaluating the impact of coherence protocols on parallel logic programming systems. In *Proceedings of the 5th EUROMICRO Workshop on Parallel and Distributed Processing*, pages 376–381, 1997. Also available as technical report ES-389/96, COPPE/Systems Engineering, May, 1996.
- [17] V. Santos Costa and Bianchini R. Optimising Parallel Logic Programming Systems for Scalable Machines. In *Proceedings of the EUROPAR'98*, Sep 1998.
- [18] V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 83–93. ACM press, April 1991. SIGPLAN Notices vol 26(7), July 1991.
- [19] Evan Tick. *Memory Performance of Prolog Architectures*. Kluwer Academic Publishers, Norwell, MA 02061, 1987.
- [20] J. E. Veenstra and R. J. Fowler. Mint: A front end for efficient simulation of shared-memory multiprocessors. In *Proceedings of the 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, 1994.

Or-Parallelism within Tabling

Ricardo Rocha, Fernando Silva, and Vítor Santos Costa*

DCC-FC & LIACC, University of Porto,
Rua do Campo Alegre, 823 - 4150 Porto, Portugal
`{ricroc,fds,vsc}@ncc.up.pt`

Abstract. One important advantage of logic programming is that it allows the implicit exploitation of parallelism. Towards this goal, we suggest that or-parallelism can be efficiently exploited in tabling systems and propose two alternative approaches, Or-Parallelism within Tabling (OPT) and Tabling within Or-Parallelism (TOP).

We concentrate on the fundamental concepts of an environment copying based model to implement the OPT approach and introduce the data structures and algorithms necessary to extend the YapOr Or-Parallel system, in order to obtain a parallel tabling system.

Keywords: Parallel Logic Programming, Or-Parallelism, Tabling.

1 Introduction

Prolog is an extremely popular and powerful logic programming language. Prolog execution is based on SLD resolution for Horn clauses. This strategy allows efficient implementation, but suffers from fundamental limitations, such as in dealing with infinite loops and redundant subcomputations. SLG resolution [3] is a tabling based method of resolution that overcomes some limitations of traditional Prolog. The method evaluates programs by storing newly found answers of current subgoals in a table. The method then uses this table to verify for repeated subgoals. Whenever such a repeated subgoal is found, the subgoal's answers are recalled from the table instead of being resolved against the program clauses. SLG resolution can thus reduce the search space for logic programs and in fact it has been proven that it can avoid looping and thus terminate for all programs that construct bounded depth terms. The XSB-Prolog system [11] was the first Prolog system to implement SLG resolution.

One important advantage of logic programming is that it allows the implicit exploitation of parallelism. This is true for SLD-based systems, and should also apply for SLG-based systems. A first proposal on how to exploit implicit parallelism in tabling systems was Freire's table-parallelism [6]. In this model, each tabled subgoal is associated with a new computational thread, a *generator thread*, that will produce and add the answers into the table. Threads that call a tabled

* The first author is thankful to PRAXIS and Fundação para a Ciência e Tecnologia for their financial support. This work was partially funded by the Melodia project (JNICT PBIC/C/TIT/2495/95).

subgoal will asynchronously consume answers as they are added to the table. This model is limitative in that it restricts exploitation of parallelism to just one implicit form of parallelism present in logic programs. Ideally, we would like to exploit maximum parallelism and take maximum advantage of current technology for parallel and tabling systems.

We observe that tabling is still about exploiting alternatives to find solutions for goals, and that or-parallel systems have precisely been designed to achieve this goal efficiently. We therefore propose two computational models, the OPT and TOP models, that combine or-parallelism and tabling by considering all open alternatives to subgoals as being amenable to parallel exploitation, be they from tabled or non-tabled subgoals. The OPT approach considers that tabling is the base component of the system, this is, each worker can be considered like a full sequential tabling engine. The or-parallel component of the system is only triggered when a worker runs out of alternatives to exploit. The TOP model unifies or-parallel suspension and suspension due to tabling. A suspended subgoal can wake up for several reasons, such as new alternatives having been found for the subgoal, the subgoal becoming leftmost, or just for lack of non speculative work in the search tree. The TOP approach considers that each worker can be considered like a sequential WAM engine, hence only managing a logical branch of the search tree, and not several branches.

In this paper we compare both models and focus on the design and implementation of the OPT model for combining or-parallelism and tabling. We chose the OPT model mainly because we believe it gives the highest degree of orthogonality between or-parallelism and tabling, thus simplifying initial implementation issues. The implementation framework is based on YapOr [9], an or-parallel system that extends Yap's sequential execution model to exploit implicit or-parallelism in Prolog programs. YapOr is based on the environment copy model, as first implemented in Muse [1]. We describe the main issues that arise in supporting tabling and its parallelization through copying. The implementation of tabling is largely based on the XSB engine, the SLG-WAM, however substantial differences exist in particular on the algorithms for restoring a computation, determining the leader node and completion operation. All these and the extended data structures already take into account the support of combined tabling and or-parallelism.

In summary, we briefly introduce the implementation of tabling. Next, we discuss the fundamental issues in supporting or-parallelism for SLG resolution and propose two alternative approaches. Then, we present the new data structures and algorithms required to extend YapOr to support tabling and to allow for a combined exploitation of or-parallelism and tabling. This corresponds to the implementation work already done and terminate by outlining some conclusions.

2 Tabling Concepts and the SLG-WAM

The SLG evaluation process can be modeled by a *SLG-forest* [10]. Whenever a tabled subgoal S is called for the first time, a new tree with root S is added to the

SLG-forest. Simultaneously, an entry for S is allocated in the *table space*. This entry will collect all the answers generated for S . Repeated calls to variants of S are resolved by consuming the answers already stored in the table. Meanwhile, as new answers are generated to S , they are inserted into the table and returned to all variant subgoals. Within this model, the nodes in the search tree are classified as either *generator* nodes, that is, first calls to tabled subgoals, *consumer* nodes, that consumer answers from the table space, and *interior* nodes, that are evaluated by the standard SLD resolution.

Space for a tree can be reclaimed when its root subgoal has been *completely evaluated*. A subgoal is said to be completely evaluated when all possible resolutions for it have been made, that is, when no more answers can be generated and the variant subgoals have consumed all the available answers. Note that a number of root subgoals may be mutually dependent, forming a *strongly connected component* (or *SCC*), and therefore can only be completed together. The completion operation is thus performed by the *leader* of the SCC, that is, by the oldest root subgoal in the SCC, when all possible resolutions have been made for all root subgoals in the SCC. Hence, in order to efficiently evaluate programs one needs an efficient and dynamic detection scheme to determine when both the subgoals in an SCC and the SCC itself have been completely evaluated.

For definite programs, tabling based evaluation has four main types of operations: (i) *Tabled Subgoal Call*, creates a generator node; (ii) *New Answer*, verifies whether a newly generated answer is already in the table, and if not, inserts it; (iii) *Answer Return*, consumes an answer from the table; and *Completion* determines whether an SCC is completely evaluated, and if not, schedules a possible resolution to continue the execution.

The implementation of tabling in XSB was attained by extending the WAM into the SLG-WAM, with minimal overhead. In short, the SLG-WAM introduces special instructions to deal with the operations above and two new memory areas: a table space, used to save the answers for tabled subgoals; and a *completion stack*, used to detect when a set of subgoals is completely evaluated.

Further, whenever a consumer node gets to a point in which it has consumed all available answers, but the correspondent tabled subgoal has not yet completed and new answers may still be generated, it must suspend its computation. In the SLG-WAM the suspension mechanism is implemented through a new set of registers, the *freeze registers*, which freeze the WAM stacks at the suspension point and prevent all data belonging to the suspended branch from being erased. A suspended consumer is resumed by restoring the registers saved in the corresponding node and by using an extension of the standard trail, the *forward trail*, to restore the bindings of the suspended branch. The resume operation is implemented by setting the `failure_continuation` field of the consumer nodes to a special `answer_return` instruction. This instruction is responsible for resuming the computation, guaranteeing that all answers are given once and just once to every variant subgoal. Through failure and backtracking to a consumer node, the `answer_return` instruction gets executed and resuming takes place.

It is upon the leader of an SCC to detect its completion. This operation is executed dynamically and must be efficiently implemented in order to minimize overheads. To achieve this, the SLG-WAM sets the `failure_continuation` field of a generator node to a special `completion` instruction whenever it resolves the last applicable program clause for the correspondent subgoal. The `completion` instruction thus ensures the total and correct evaluation of the subgoal search space. This instruction is executed through backtracking. In the default XSB scheduling strategy (*batched scheduling* [7]), it resumes the consumer node corresponding to the deeper variant subgoal with unconsumed answers. The computation will consume all the newly found answers, backtrack to other consumer nodes (higher up in the chain of consumer nodes) with unconsumed answers, and fail to the generator node, until no more answers are left to consume. At this point, if that node is the leader of its SCC, a fixpoint is reached, all dependent subgoals are completed, and the subgoal can be marked completed. Otherwise, the computation will fail back to the previous node and the fixpoint check will later be executed in an upper generator node.

3 Parallel Execution of Tabled Programs

Ideally, we would like to exploit maximum parallelism and take maximum advantage of current technology for tabling systems. We propose that all alternatives to subgoals should be amenable to parallel exploitation, be they from normal or tabled subgoals, and that or-parallel frameworks can be used as the basis to do so. This gives an unified approach with two major advantages. First, it does not restrict parallelism to tabled subgoals and, second, it can draw from the very successful body of work in implementing or-parallel systems. We believe that this approach can result in efficient models for the exploitation of parallelism in tabling-based systems. We envisage two different models to combine or-parallelism and tabling:

Or-Parallelism within Tabling (OPT) In this approach, parallelism is exploited between independent tabling engines, that share alternatives. Tabling is the base component of the system, this is, each worker can be considered a full sequential tabling engine and should spend most of its computation time exploiting the search tree involved in such an evaluation. It thus can allocate all three types of nodes, fully implement suspension of tabled subgoals, and resume subcomputations to consume newly found answers. Or-parallelism is triggered when a worker runs out of alternatives to exploit. In the OPT approach, unexploited alternatives should be made available for parallel execution, regardless of whether they originate from a generator, consumer or interior node. Therefore, parallelism can and should stem from both tabled and non-tabled subgoals.

From the viewpoint of or-parallelism, the OPT approach generalizes Warren's multi-sequential engine framework for the exploitation of or-parallelism. Or-parallelism now stems from having several engines that implement SLG-resolution, instead of implementing Prolog's SLD-resolution.

Fig. 1 gives an example of this approach for the following small program and the query $?- a(X)$. We assume two workers, W1 and W2.

```

:- table a/1.          b(1).
a(X) :- a(X).          b(X) :- ...
a(X) :- b(X).          b(X) :- ...

```

Consider that worker W1 executes the query goal. It first inserts an entry for the tabled subgoal $a(X)$ into the table and creates a generator node for it. The execution of the first alternative leads to a recursive call for $a/1$, thus the worker creates a consumer node for $a/1$ and backtracks. The next alternative finds a non-tabled subgoal $b/1$ for which an interior node is created. The first alternative for $b/1$ succeeds and an answer for $a(X)$ is therefore found ($a(1)$). The worker inserts the newly found answer in the table and then starts exploiting the next alternative of $b/1$.

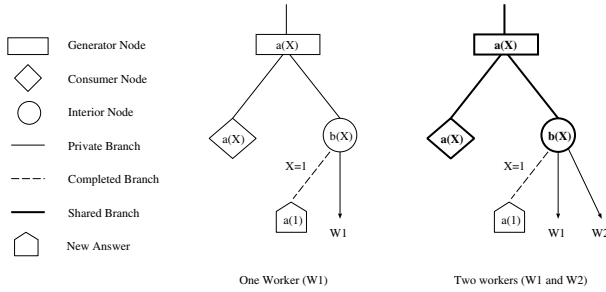


Fig. 1. Sharing work in a SLG tree.

At this point, worker W2 moves in to share work. Consider that worker W1 decides to share work up to its last private node (that is, the interior node for $b/1$). The two workers will share three nodes: the generator node for $a/1$, the consumer node for $a/1$ and the interior node for $b/1$. Worker W2 takes the next unexploited alternative of $b/1$ and from now on, both workers can quickly find further answers for $a(X)$ and any of them can restart the shared consumer node.

Tabling Unified with Or-Parallelism (TOP) We have seen that in tabling based systems subgoals need to suspend on other subgoals obtaining the full set of answers. Or-parallel systems also need to suspend, either while waiting for left-mostness in the case of side-effects, or to avoid speculative execution. The need for suspending introduces an important similarity between tabling and or-parallelism. The TOP approach unifies or-parallel suspensions and suspensions due to tabling. When exploiting parallelism between branches in the search tree, some branches may be suspended, say, because they are speculative or not left-most nodes, or because they are consumer nodes waiting for more solutions, while others are available for parallel execution.

Fig. 2 shows parallel execution for the previous program under this approach. The left figure shows that as soon as W1 suspends on consumer node for $a/1$, it makes the current branch of the search tree public and backtracks to the upper node. The consumer node for $a/1$ can only be resumed after answers to $a(X)$ are found. In the left figure an answer for subgoal $a(X)$ was found. So, worker W2 can choose whether to resume the consumer node with the newly found answer or to ask worker W1 to share his private nodes. In this case we represent the situation where worker W2 resumes the consumer node.

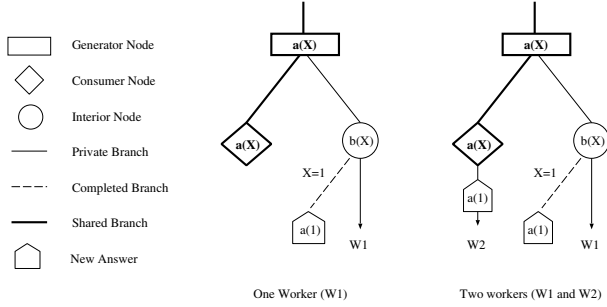


Fig. 2. Unifying suspension in parallelized tabling execution.

Comparing the Two Models The TOP model is a very attractive model, as it provides a clean and systematic unification of tabling and or-parallel suspensions. Workers have a clearly defined position, because a worker always occupies the tip of a single branch in the search tree. Everything else is shared work. It also has practical advantages, such as the fact that in this approach we can guarantee a suspended branch will only appear once, instead of possibly several times for several workers.

On the other hand, as suspended nodes are always shared in or-parallel systems, the unified suspension may result in having a larger public part of the tree which may increase overheads. Besides, to support all forms of suspension with minimal overhead, the unified suspension mechanism must be implemented efficiently. Moreover, support for the tabling component in the TOP approach requires a slightly different tabling engine than SLG-WAM. The recently proposed CAT model [4] seems to fulfill best the requirements of the TOP approach, since it assumes a linear stack for the current branch and uses an auxiliary area to save the suspended nodes. Therefore, in order to implement the TOP approach using CAT, we should adopt, for the or-parallel component, an environment copying model (such as used in Muse) as it fits best with the kind of operations CAT introduces.

In this regard, the OPT approach offers some interesting advantages. It enables different combinations for or-parallelism and tabling, giving implementors the highest degree of freedom. For instance, one can use the SLG-WAM for

tabling, and environment copying [1] or binding arrays [5] for or-parallelism. Moreover, the OPT approach reduces to a minimum the overlap between or-parallelism and tabling, as we only have a tabling system extended with an or-parallel component. In TOP, we have a standard Prolog system extended with a tabling/or-parallel component.

In this work, we focus on the design and implementation of the OPT approach, adopting the SLG-WAM for tabling and environment copying for or-parallelism. Our choice seems the most natural as we believe the OPT approach gives the highest degree of orthogonality between or-parallelism and tabling. The hierarchy of or-parallelism within tabling results in a property that one can take advantage of to structure, and thus simplify, scheduler design and initial implementation issues.

Overview of the Model In our model, a set of workers, will execute a tabled program by traversing its search tree, whose nodes are entry points for parallelism. Each worker physically owns a copy of the environment (that is, the stacks) and shares a large area related to tabling and scheduling. During execution, the search tree is implicitly divided in public and private regions. Workers in the private region execute nearly as in sequential tabling. A worker with excess of work (that is, with private nodes with unexploited alternatives or unconsumed answers) when prompted for work by other workers, shares some of their private nodes. When a worker shares work with another worker, the incremental copy technique is used to set the environment for the requesting worker. Whenever a worker backtracks to a public node it synchronizes to perform the usual actions that are executed in sequential tabling. For the generator and interior nodes it takes the next alternative, and for the consumer nodes it takes the next unconsumed answer. If there are no alternatives or no unconsumed answers left, then the worker executes the *public completion*¹ operation.

4 Extending YapOr to Support Tabling

We next discuss the main data structures to extend YapOr to support parallel tabling. In the initial design, we will only consider table predicates without any kind of negative calls.

Data Areas The data areas necessary to implement the complete or-parallel tabling system are shown in the memory layout depicted in Fig. 3. The memory is divided into a global shared area and into a number of logically private areas, each owned by a single *worker* in the system. The private areas contains the standard WAM stacks, as required for each worker. The global shared area includes four main sub-areas, that we describe next.

The *or-frame space* is required by the or-parallel component in order to synchronize access to shared nodes [1] and to store scheduling information. The *table space* is required by the tabling component. It contains the table structure

¹ A public completion operation is a completion operation executed in a shared node.

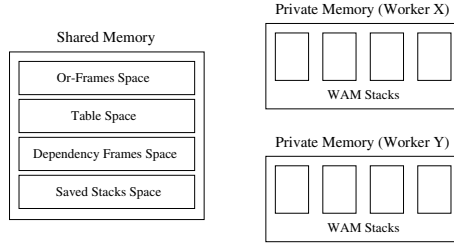


Fig. 3. Memory layout for the proposed OPT approach.

and has to be stored in shared memory for fast access to tabled subgoals by all the workers.

The *dependency frames space* is a novel data structure, designed to support or-parallel tabling. Dependency frames extend consumer nodes by holding supplementary information about a suspension point. They must be implemented in shared memory in order to permit synchronization among workers in the management of the suspension points, and in order to avoid unnecessary copying of information when a consumer node is shared between workers. This data structure is discussed in more detail below.

The *saved stacks space* preserves stacks of suspended branches. This space is required by a purely or-parallel implementation to save worker’s suspension on builtins that require to be leftmost, or *voluntarily* abandon speculative work in favor of work that is not likely to be pruned [2]. In both cases, workers suspend their current work by copying their stacks into the saved stacks space, and start searching for other work. When we execute tabling in parallel, workers that share nodes can have dependencies between them. Thus, sometimes it may be necessary to delay the execution of a public completion operation until no more dependencies exist. On the other hand, in order to allow the parallel exploitation of other alternatives, as required by the environment copy model, it is necessary to maintain the stacks coherent with those of the workers that share the same nodes. These two objectives can be achieved by saving in the saved stacks space the parts of the stacks that correspond to the branches in the range of the public completion operation.

Table Space The design and implementation of the data structures and algorithms to efficiently access the table space is one of the critical issues in the design of a tabling system. Next, we provide a brief description of our implementation. It uses tries as the basis for tables as proposed in [8]. Tries provide complete discrimination for terms and permit a lookup and possible insertion to be performed in a single pass through a term. Tries are also easily parallelizable.

The table space can be accessed in different ways during the course of an evaluation: to look up if a subgoal is in the table, and if not insert it; to verify whether a newly found answer is already in the table, and if not insert it; to pick up answers from the table to consumer nodes; and finally to mark subgoals as completed during a completion operation.

Fig. 4 presents the table data structures for a particular predicate `t/2` after the execution of some `tabled_subgoal_call` and `new_answer` instructions.

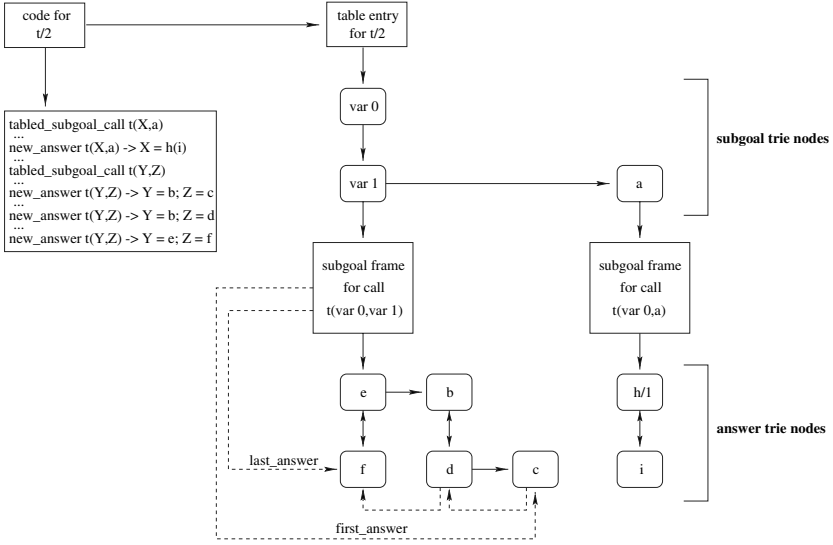


Fig. 4. Using tries to organize the table space.

The subgoal frames delimit the subgoal and answer trie nodes space. They are used to easily access the answers found for a particular subgoal and to efficiently check if new answers have been found for a particular consumer node, through its `last_answer` pointer. The subgoal frames are also used by the completion operation to mark a subgoal as completed.

Each invocation of the `tabled_subgoal_call` instruction leads to either finding a path through the subgoal trie nodes, always starting from the table entry, until reaching a matching subgoal frame, or to creating the correspondent path of subgoal trie nodes, otherwise. Each invocation of the `new_answer` instruction corresponds to the definition of a path through the answer trie nodes, starting from the corresponding subgoal frame. In the example, the subgoal trie node with value `var 0` and the answer trie nodes with value `b` belong to several paths. This design property merges the paths that have the same first arguments. Note also that each trie node can only hold atoms, variables, or functors. Thus we need two nodes to represent the answer `h(i)`, one for the functor `h/1` and the other for the argument `i`.

Accessing and updating the table space must be carefully controlled in a parallel system. We want to maximize parallelism, whilst minimizing overheads. Read/write locks are the ideal data structure for this purpose. One can have a single lock for the table, thus only enabling a single writer for the whole table, one lock per table entry allowing one writer per procedure, one lock per path

allowing one writer per call, or one lock per every trie node to attain most parallelism. Experimental evaluation is required to find the best solution.

Dependency Frames The dependency frame is the key data structure required to control suspension, resumption and completion of subcomputations. This data structure serves to: save information about the suspension point; connect consumer nodes with the table space, to search for and to pick up new answers; and form a dependency graph between consumer nodes, to efficiently check for leader nodes and perform completion.

Fig. 5 shows an example of an evaluation involving dependency frames. The sub-figure to the left presents the dependencies between the predicates involved.

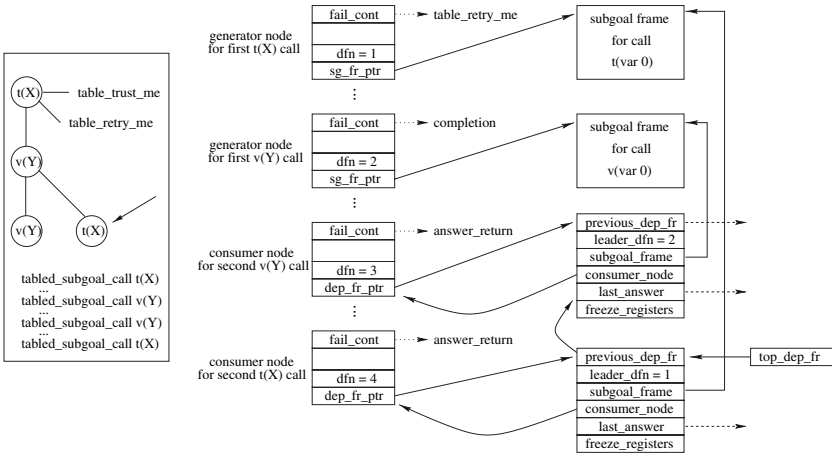


Fig. 5. Table data structures dependencies.

The first instance of `tabled_subgoal_call` searches the table space for the corresponding subgoal $t(X)$. Supposing this is the first call to the subgoal, it must allocate a subgoal frame and create a generator choice point. In our scheme, a generator choice point is simply a standard choice point plus a new `sg_fr_ptr` field, pointing to the newly allocated subgoal frame. The pointer is used to add new solutions and to check for completion. In order to mark the precedence between choice points in the left-to-right, top-down search, we introduce an extra field, `dfn` (depth first number), in all choice points. This field is not strictly necessary in the implementation because we can achieve the same goal by using the addresses of the choice points, however it simplifies the description of the model.

Following the example, an analogous situation occurs with the first call to subgoal $v(Y)$. The repeated call to subgoal $v(Y)$ allocates a dependency frame and creates a consumer choice point. A consumer choice point is a standard

choice point plus an extra `dep_fr_ptr` field, pointing to the newly allocated dependency frame.

A dependency frame contains six fields. The `previous_dep_fr` field points to the previous allocated dependency frame. The last dependency frame is pointed by the global variable `top_dep_fr`. The `leader_dfn` field stores the `dfn` of the bottommost leader node (details are presented next). The `subgoal_frame` field is a pointer to the correspondent subgoal frame. The `consumer_node` field is a back pointer to the consumer node. The `last_answer` field is a pointer to the last consumed answer, and is used in conjunction with the `subgoal_frame` field to check if new answers have been found for the subgoal call. The `freeze_registers` fields stores the current top positions of each of the stacks. When a completion instruction is executed with success, we consult the `freeze_registers` of the resulting top dependency frame to restore the top positions of each stack and release space. Note that in the SLG-WAM these registers are kept at the nodes where completion may take place, that is the generator nodes. In our case, this solution is not adequate as we may execute completion in any node.

Finally, the second call to `t(X)` implies an analogous situation to the previous one. A dependency frame and a consumer choice point are allocated and the `top_dep_fr` is updated.

5 The Flow of Control

A tabling evaluation can be seen as a sequence of suspension and resumptions of subcomputations. A computation suspends every time a consumer node has consumed all available answers and resumes when new solutions are found.

Restoring a Computation Every time a consumer node is allocated, its failure continuation pointer is made to point to an `answer_return` instruction. The instruction is executed through failure to the node, and guarantees that every answer is consumed once and just once. Before resuming a computation, it is necessary to restore the WAM registers and the variable bindings at the suspension point. The WAM register values are saved in the consumer node and the variable bindings are saved in the *forward trail*. The forward trail is an extension of the standard WAM trail that includes variable bindings in trail operations.

The SLG-WAM uses a forward trail with three fields per each frame. They record the address of the trailed variable (as the standard WAM trail), the value to which the variable was bound and a pointer to the parent trail frame which permits to chain correctly the variables through the current branch, hence jumping across the frozen segments (see [10] for more details). In our approach we only use the first two fields to implement the forward trail, thus spending less space in the trail stack. As Yap already uses the trail to store information beyond the normal variable trailing (to control dynamic predicates and multi-assignment variables), we extend this information to also control the chain between the different frozen segments. In terms of computational complexity the two approaches are equivalent. The main advantage of our scheme is that Yap already tests the

trail frames to check if they are of a special type, and so we do not introduce further overheads in the system. It would be the case if we had chosen to implement the SLG-WAM approach.

Leader Nodes In sequential tabling, only generator nodes can be leader nodes, hence only they perform completion. In our design any node can be a leader. We must therefore check for leader nodes whenever we backtrack to a private generator node or to any shared node. If the node is leader we perform completion, otherwise we simply fail and backtrack. We designed our algorithms to quickly determine whether a node is a leader.

```
find_bottommost_leader_node () {
    leader_dfn = direct_dependency_dfn();
    aux_dep_fr = top_dep_fr;
    while (aux_dep_fr != NULL &&
           leader_dfn < DependencyFrame_consumer_dfn(aux_dep_fr)) {
        if (leader_dfn >= DependencyFrame_leader_dfn(aux_dep_fr))
            return DependencyFrame_leader_dfn(aux_dep_fr);
        aux_dep_fr = DependencyFrame_previous_dep_fr(aux_dep_fr);
    }
    return leader_dfn;
}
```

Fig. 6. Pseudo-code for `find_bottommost_leader_node()`.

Fig. 6 presents the pseudo-code to initialize the `leader_dfn` field when we allocate a new dependency frame. The `direct_dependency_dfn()` function returns the `dfn` of the depth-most node that contains in one of the branches below it, the generator node of the variant subgoal call. In sequential tabling, as we have all nodes, the depth-most node that contains the generator node for a particular subgoal call, is the generator node itself. Hence, the `dfn` returned is the value of the `dfn` field within the generator node. Fig. 7 illustrates the dependencies between the several depth first numbers involved in the `find_bottommost_leader_node()` algorithm, in the case of a parallel evaluation. In order for a worker to test if a certain node, of its own branch, is leader or not, it has to check if the `dfn` field of the node is equal to the `leader_dfn` field found in the `top_dep_fr` register. In Fig. 7, workers 1 and 2 have leader nodes at generator node **a** and interior node **b** respectively.

Public Completion The correctness and efficiency of the completion algorithm appears to be one of the most important points in the implementation of a combined tabling/or-parallel system. Next we present a design for our OPT-based implementation.

If a leader node is shared and contains consumer nodes below it, this means that it depends on branches explored by other workers. Thus, even after a worker having backtracked to a leader node, it may not execute the completion operation

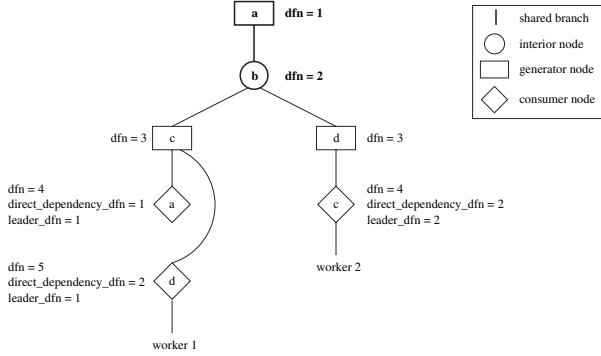


Fig. 7. *Depth first number dependencies.*

immediately. The reason for this is that the other workers can still influence the leader branch. As a result, it becomes necessary to suspend² the leader branch, and therefore, allow the current worker to continue execution. This suspension mechanism includes, saving the correspondent part of the stacks to the *save stacks space* (putting the respective reference in the leader node) and readjusting the freeze registers. The save stacks space resides in a shared area to allow any other worker to complete a suspended branch. This scheme also enables the current worker to proceed its execution. If the worker would not suspend the leader branch, hence not saving the stacks in the shared space, then a future sharing work operation would damage the stack areas related to the leader branch and therefore would make the completion operation unworkable. An alternative would be for the worker responsible for the leader branch to wait until no one else could influence it and only then complete the branch. Obviously, this is not an efficient strategy and besides it may carry us into a deadlock situation.

Fig. 8 shows the pseudo-code for the `public_completion()` operation. This operation is executed when a worker backtracks to a shared interior/generator node with no more alternatives left, or to a shared consumer node without unconsumed answers. The last worker leaving a node is responsible for checking and collecting all the suspension branches that have unconsumed answers. To resume a suspension branch a worker needs to copy the saved stacks to the correct position in its own stacks. Thus, for a worker to resume immediately a suspension branch, it has first to suspend its current branch and only later restart it. This has the disadvantage that the worker has to make two suspensions and resumptions instead of just one. Hence, we adopted the strategy of resuming the collected branches only when the worker finds itself in a leader node position. Here, a worker either completes the correspondent branch or suspends it. In both

² The notion of suspension in this context is obviously different from the one presented for tabling.

```

public_completion (node N) {
  if (last worker in node N)
    for all suspension branches SB stored in node N
      if (exists unconsumed answers for any consumer node in SB)
        collect (SB) /* to be resumed later */
  if (N is a leader node)
    if (exists unconsumed answers for any consumer node below node N)
      backtrack_through_new_answers() /* as in SLG-WAM */
    if (suspension branches collected)
      suspend_current_branch()
      resume (a suspension branch)
    else if (not last worker in node N)
      suspend_current_branch()
    else if (hidden workers in node N)
      suspend_current_branch()
    else
      complete_all()
  else /* not leader */
    if (consumer nodes below node N)
      increment hidden workers in node N
  backtrack
}

```

Fig. 8. Pseudo-code for `public_completion()`.

situations, the stacks do not contain frozen segments below the leader node and therefore we do not have to pay extra overheads to resume a collected branch.

Whenever a node finds that it is a leader, it starts to check if there are no consumer nodes with unconsumed answers below. If there is such a node, it resumes the computation to the deeper consumer node with unconsumed answers. To check if there is a node with unconsumed answers, we can follow the dependency frames chain corresponding to the consumer nodes below, and check for one such that the `last_answer` pointer is different from the one stored on the subgoal frame pointed by the `subgoal_frame` field.

When a worker backtracks from a node and that node stays preserved in its stacks, the worker has to increment the *hidden workers* counter of the node. This counter indicates the number of workers that are executing in upper nodes and still contain the node. These workers can influence the node, if a resume operation takes place and it includes the node.

In this public completion algorithm, a worker only completes a leader node when there is nothing that can influence its branches. This only happens, when there are no suspended branches collected and there are no workers in the node, be they physically present or hidden. Completing a node includes marking the tabled subgoals involved as completed, releasing memory space and readjusting the freeze registers.

The public completion scheme proposed has two major advantages. One is that there is no communication or explicit synchronization between workers, therefore it reduces significantly possible overheads. The second advantage is that the leader nodes are the only points where we suspend branches. This is very important, since it reduces the number of check points we have to make

in order to ensure that all answers are consumed in a suspended branch, to just one. This check point is executed by the last worker that leaves a node. Besides, in upper nodes we do not need to care about the uncollected suspended branches stored in the bottom nodes because we know that no upper branches can influence them.

6 Conclusions

In this paper we presented two approaches to combine or-parallelism and tabling, and focused on the design of data structures and algorithms to implement the OPT approach. These new structures and algorithms were built from within the environment copying based system, YapOr, in order to obtain a parallel tabling system. Currently, we have sequential tabling and or-parallelism functioning separately within the same system. However, we already have in the system all the data structures necessary to combine their execution and conforming with the description given in this paper. We are now working on adjusting the system for parallel execution of tabling. This will require changes to the current scheduler in order to efficiently and correctly execute tabled logic programs in parallel.

References

- [1] K. A. M. Ali and R. Karlsson. Full Prolog and Scheduling OR-Parallelism in Muse. *Intern. Journal of Parallel Programming*, 19(6), Dec. 1990.
- [2] K. A. M. Ali and R. Karlsson. Scheduling Speculative Work in Muse and Performance Results. *Intern. Journal of Parallel Programming*, 21(6), Dec. 1992.
- [3] W. Chen and D. S. Warren. Query Evaluation under the Well Founded Semantics. In *Proc. of PODS'93*, pages 168–179, 1993.
- [4] B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. In *Proc. of PLILP/ALP'98*. Springer-Verlag, Sep. 1998.
- [5] E. Lusk et. al. The Aurora Or-parallel Prolog System. In *Proc. of FGCS'88*, pages 819–830. ICOT, Nov. 1988.
- [6] J. Freire, R. Hu, T. Swift, and D. S. Warren. Exploiting Parallelism in Tabled Evaluations. In *Proc. of PLILP'95*, pages 115–132. Springer-Verlag, 1995.
- [7] J. Freire, T. Swift, and D. S. Warren. Beyond Depth-First: Improving Tabled Logic Programs through Alternative Scheduling Strategies. In *Proc. of PLILP'96*, pages 243–258. Springer-Verlag, Sep. 1996.
- [8] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient Tabling Mechanisms for Logic Programs. In *Proc. of ICLP'95*, pages 687–711. The MIT Press, June 1995.
- [9] R. Rocha, F. Silva, and V. S. Costa. YapOr: an Or-Parallel Prolog System based on Environment Copying. Technical Report DCC-97-14, DCC-FC & LIACC, University of Porto, Dec. 1997.
- [10] K. Sagonas and T. Swift. An Abstract Machine for Tabled Execution of Fixed-Order Stratified Logic Programs. *Journal of ACM Transactions on Programming Languages and Systems*, 1998. To appear.
- [11] K. Sagonas, T. Swift, and D. S. Warren. XSB as an Efficient Deductive Database Engine. In *Proc. of ACM SIGMOD International Conference on the Management of Data*, pages 442–453, May 1994.

Mnesia - A Distributed Robust DBMS for Telecommunications Applications

Håkan Mattsson, Hans Nilsson, and Claes Wikström

Computer Science Laboratory

Ericsson Telecom AB

S-125 26 Stockholm, Sweden

{hakan,hans,klacke}@erix.ericsson.se

Abstract. The Mnesia DBMS runs in the same address space as the application owning the data, yet the application cannot destroy the contents of the data base. This provides for both fast accesses and efficient fault tolerance, normally conflicting requirements. The implementation is based on features in the Erlang programming language, in which Mnesia is embedded.

1 Introduction

The management of data in telecommunications system has many aspects whereof some, but not all, are addressed by traditional commercial DBMSs (Data Base Management Systems). In particular the very high level of fault tolerance which is required in many nonstop systems, combined with requirements on the DBMS to run in the same address space as the application, have led us to implement a brand new DBMS. This paper describes the motivation for, as well as the design of this new DBMS, called Mnesia. Mnesia is implemented in, and very tightly connected to, the programming language Erlang and it provides the functionality that is necessary for the implementation of fault tolerant telecommunications systems. Mnesia is a multiuser Distributed DBMS specially made for industrial telecommunications applications written in the symbolic programming language Erlang [1] which is also the intended target language. Mnesia tries to address all of the data management issues required for typical telecommunications systems and it has a number of features that are not normally found in traditional databases.

In telecommunications applications there are needs different from the features provided by traditional DBMSs. The applications we now see implemented in the Erlang language need a mixture of a broad range of features which generally are not satisfied by traditional DBMSs. Mnesia is designed with requirements like the following in mind:

1. Fast realtime key/value lookup.
2. Complicated non realtime queries mainly for operation and maintenance.
3. Distributed data due to distributed applications.
4. High fault tolerance.

5. Dynamic re configuration.
6. Complex objects.

What sets Mnesia apart from most other DBMSs is that it is designed with the typical data management problems of telecommunications applications in mind. Hence Mnesia combines many concepts found in traditional databases such as transactions and queries with concepts found in data management systems for telecommunications applications such as very fast realtime operations, configurable degree of fault tolerance (by means of replication) and the ability to reconfigure the system without stopping or suspending it. Mnesia is also interesting due to its tight coupling to the programming language Erlang, thus almost turning Erlang into a database programming language. This has many benefits, the foremost being that the impedance mismatch [5] between data format used by the DBMS and data format used by the programming language which is being used to manipulate the data, completely disappears.

Mnesia is currently used in almost all Erlang based projects within Ericsson ranging from small scale prototype projects to major switching product projects.

The remainder of this paper is organized as follows. Section 2 is a brief overview of the DBMS. Section 3 is organized as a listing of typical DBMS functionalities, a discussion of some telecommunications aspect on the functionality and how the functionality is provided by Mnesia. Section 4 contains some performance measurements and finally section 5 provides some conclusions.

2 A Brief Overview of Mnesia

We briefly overview the features of Mnesia DBMS. Mnesia is both an extension of the programming language Erlang as well as an Erlang application. The DBMS components such as lock manager, transaction manager, replication manager, logger, primary and secondary memory storage, backup system, etc are all implemented as regular Erlang programs. The query language, however, is implemented as a part of the actual Erlang syntax, whereas the optimizing query compiler and evaluator are regular Erlang programs. The data model of Mnesia is of a hybrid type: data is organized as tables of records similar to relations, but the attributes of the records (including the key) can hold arbitrarily complex compound data structures such as trees, functions, closures, code etc. As such, Mnesia could be characterized as a so called object-relational DBMS. Let us assume, a definition for a person record.

```
-record(person, {name,          %% atomic, unique key
                 data,          %% compound unspecified structure
                 married_to,    %% name of partner or undefined
                 children}).    %% list of children
```

Given this record definition, it is possible to create instances of the person record with the following Erlang syntax where

```
X = #person{name = klacke,
             data = {male, 36, 971191},
             married_to = eva,
             children = [marten, maja, klara]}.
```

binds the variable `X` to a new person record. The data field is bound to a tuple `{male, 36, 971191}` with three parts. This is an example of a complex object, and Mnesia puts no constraints on the complexity which is allowed on attributes. We can even have function objects with variable closures as attribute values. The variable `X` is merely an Erlang term, and in order to insert it into the database, it must be written:

```
mnesia:write(X)
```

Series of Mnesia operations can be performed together as an atomic transaction. In order to have Mnesia execute a transaction, the programmer must construct a functional object and then present the Mnesia system with that functional object similar to [7]. We explain this by an example, assume we wish to write an Erlang function `divorce(Name)` which takes the name of a person, finds the person from the database, and sets the `married_to` field for the person and its partner back to the undefined value.

```
divorce(Name) ->
  F = fun() >
    case mnesia:read(Name) of
      [] >
        mnesia:abort(no_such_person);
      Pers >
        Partner = mnesia:read(Pers#person.married_to),
        mnesia:write(Pers#person{married_to = undefined}),
        mnesia:write(Partner#person{married_to = undefined})
    end
  end,
  mnesia:transaction(F).
```

The `divorce/1` function consists of two statements, the first `F =` statement creates a function object, it does not execute anything, it merely constructs an anonymous function. The second statement gives this function away to the Mnesia system which executes the function in the context of a transaction, adhering to traditional transaction semantics [2].

The actual function `F` first executes a read operation to find the person with the name `Name`, it then executes a second read to find the partner of the first person, and finally it writes two new records into the database with the `married_to` field set to undefined. These two write operations will effectively overwrite the old values. The function `divorce/1` returns the value of the transaction, which is either `{aborted, Reason}` or `{atomic, Value}` depending on whether the transaction was aborted or not.

Queries in Mnesia are expressed with a list comprehension syntax [15]. A query to find the names of all persons with more than X children is formulated as:

```
query [P.name || P <- table(person),
      length(P.children) > X]
end
```

This should be read as: construct the list of P.name's such that P is taken from the table of persons, and length of the children list is greater than X. It is indeed both possible and natural to mix a query with user defined predicates. Assume a function:

```
maturep({Sex, Age, Phone}) when Age > 30 >
  true;
maturep({Sex, Age, Phone}) >
  false.
```

Then the query:

```
query [P.name || P <- table(person),
      maturep(P.data),
      length(P.children) > X]
end
```

extracts all persons with more than X children and where the second element of the data field is larger than 30. It is also possible to define rules with an embedded logic language similar to Datalog [16]. If we define the rule:

```
oldies(Name) :-
  P <- table(person),
  maturep(P.data),
  Name = P.name.
```

This defines a rule, which then acts as a virtual table and application programs can access the virtual table oldies. The virtual oldies table contains a subset of the real person table. This is similar to, but more powerful than, the concept of views found in relational databases. Queries are compiled by an optimizing query compiler which has been integrated with the normal Erlang compiler.

Tables can be replicated to several sites (or nodes). The network of nodes can be a heterogeneous network. Replication is the mechanism whereby we can construct fault tolerant systems. Access to a table is location transparent, that is, programs do not have to have explicit knowledge of the location of data. A table has a unique name and certain properties, we have the following list of properties attached to each table.

- *type* controls whether the table has set or bag semantics. A set has unique keys, whereas a bag can have several objects with the same key.
- *ram_copies* a list of Mnesia nodes where replicas of the table are held in ram only.
- *disc_copies* a list of Mnesia nodes where replicas of the table are held entirely in ram, but all update operations on the table are logged to disc.
- *disc_only_copies* a list of Mnesia nodes where replicas of the table are held on disc only. These replicas are of course considerably slower than replicas held in ram.
- *index* a list specifying on which attributes of the record index information shall be maintained. All records are always automatically indexed on the primary key.
- *snmp* controls whether the table shall be possible to manipulate through the Simple Network Management Protocol protocol [4].

Descriptions of all tables are kept in a database schema and Mnesia has a multitude of functions to manipulate the schema dynamically. Tables can be created, moved, replicated, changed, destroyed etc. Furthermore all these system activities are performed in the background and thus allows the application to utilize the system as usual although the system itself is being changed.

Backups can be constructed of the entire distributed system, these backups can be installed as fallbacks. This means that if the system should crash, the database will automatically be recreated from the fallback.

3 DBMS Feature Discussion

Different DBMSs have different features and characteristics. This section is organized as a listing of different DBMS characteristics and a short discussion of the importance and necessity in our intended application domain.

3.1 Complex Values

The ability to handle complex values, such as lists, sets, trees, etc efficiently and naturally in the DBMS is probably the single most important feature for a telecommunications DBMS. Telecommunications applications which handle traffic are usually driven by external stimuli which arrives at the system. When such a stimuli arrives in the form of a PDU (Protocol Data Unit) to the system, the PDU is decoded and some appropriate action must be taken. When the PDU has been decoded, the system usually has to retrieve some data object, possibly a subscriber record which is used to decide what action should be taken in response to the received PDU. In many telecommunications systems the single most important feature of the data management system is that this lookup operation is very efficient. It is also important that the DBMS allows the data to be structured and stored in such a way so that relevant data can be accessed in a single lookup operation. This fact makes it hard to model telecommunications

system with traditional relational databases. Organizing the telecommunications data in third (or even first) normal form is usually not possible.

This is one of the reasons why the telecommunications industry have paid so much attention to object-oriented database systems which allows the data to be organized in a more flexible way than relational database systems. Thus Mnesia allows the user to use arbitrarily complex objects both as attribute values but also as key values in the database.

3.2 Data Format and Address Space

Many databases use an internal, language independent format to store data. This is most unfortunate in telecommunications systems due to the previously mentioned fast lookup requirement. Many object oriented DBMSs are tightly coupled to a programming language like C++ or Smalltalk. The ability to manipulate regular programming language objects in the database, makes the impedance mismatch disappear. This not only eases the manipulation of the DBMS but it also provides an opportunity to implement very efficient lookup operations since a lookup can immediately return a pointer to the object by means of the programming language being used. If we for example want to implement a routing table as a database table, it is not realistic to transform the routing data back and forth from an external DBMS format for each packet we need to route. Furthermore it is not realistic to perform any context switches and search the relevant data for each packet in a process executing in another address space. This effectively rules out all DBMSs that cannot be directly linked into the address space of the application, as well as all DBMSs that are linked into the application but use a language independent format to store data.

The major disadvantage of letting the application run in the same address space as the DBMS is that if the application should crash due to a programming error, the DBMS may not be given the opportunity to store vital data to secondary storage before terminating. This means that the entire DBMS must be recovered before starting up again. This is usually a time consuming process and in telecommunications system, the downtime must be short. We avoid this problem since the applications as well as the DBMS are implemented in the Erlang language. An Erlang application cannot crash in such away that it effects the DBMS. The application run in the same address space as the DBMS, but Erlang itself makes it impossible for the crash of one application to effect an other application. Erlang processes have the efficiency advantage of running in the same address space but they do not have the possibility to explicitly read or write each others memory.

3.3 Fault Tolerance

Many telecommunications applications are nonstop systems. The system must be able to continue to provide its services even if a number of hardware or software errors occur. This adds requirements not only to the DBMS, but also to the

telecommunication application itself. It influences the design of the entire application and the DBMS must provide the application designers with mechanisms whereby a sufficiently fault tolerant system can be designed. The mechanism provided by Mnesia is the ability to replicate a table to several nodes. All replicas of a Mnesia table are equal and there is thus no concept of primary and standby copies at the DBMS level. If a table is replicated all write operations are applied to all replicas for each transaction. If some replicas are not available the write operations will succeed anyway and the missing replicas will be later updated when they are recovered. This mechanism makes it possible to design systems where several geographically distinct systems cooperate to provide a continuously running nonstop system. Many other highly fault tolerant systems like ClustRa [11] also provide fault tolerance through means of replication. However, they do not have the ability to execute in the same address space as the application.

Mnesia can also recover partially from complete disasters. All objects that are written to disc, are coded in such away that it is possible to safely distinguish data from garbage. This makes it possible to scan a damaged or crashed disc or filesystem and retrieve data from the crashed disc.

3.4 Distribution and Location Transparency

Mnesia is a truly distributed DBMS where data can be replicated or simply reside remotely. In such an environment it is important that the DBMS programmer can access data without explicit knowledge about the location of data. That is, location transparency of data is important. On the other hand, since it is most certainly more expensive to access data remotely, it must also be possible for the application programmer to explicitly find this location information in order to execute the program where the data is. Hence, we want to provide location transparency as well as the ability to explicitly locate data. Different applications have different requirements.

Mnesia applications which access tables by using the name of the table only, work regardless of the location of the table. The system keeps track of where data is replicated. However, it is also possible for the Mnesia programmer to query the system for the location of a table, and then execute the code remotely instead. This can be done by sending the code to the remote site or by ensuring that the code is preloaded there.

3.5 Transactions and ACID

DBMSs have ACID properties, Atomicity, Consistency, Isolation and Durability. These properties are implemented by means of transactions, writeahead logging and recovery in Mnesia. Most Mnesia transactions consists of series of operations on ram only (possibly replicated) tables. These transactions do not interact with the disc storage system at all, hence the Durability property is not fulfilled for these transactions. An example where transaction semantics are required in telecommunication systems is when we add a new subscriber to the system.

When we do this, several resources are allocated in the system, and several data objects are written into the memory of the system. It is vital that all of these operations are performed as one single atomic action. Otherwise the system could end up in an inconsistent state with possibly unreleased resources.

3.6 The Ability to Bypass the Transaction Manager

The overhead for a transaction is quite high and for certain parts of traffic handling applications, it is simply not feasible to use a transaction system to access the data. Consequently, a DBMS that is suitable for telecommunications must be able to support both atomic transactions consisting of series of database operations as well as very light weight locking on the same data. The traffic system consists of a number of tables. Many of these tables are seldom written but very often read. For example it is more common to process a single call than it is to add a subscriber and it is more common to route a PDU packet than it is to change the routing tables.

When we execute performance critical code, we do not want to impose the overhead of an entire transaction in order to read data. On the contrary, if for example packet routing code reads routing information from a routing table while the routing table is being updated, it is acceptable that some packets get lost due to this access conflict. What is needed here, is very lightweight locking protection so that application processes can access the data tables and be certain that each data object that is read, is not garbled due to concurrent writers. This is supported by Mnesia through a so called dirty interface. It is possible to read, write and search Mnesia tables without protecting the operation inside a transaction. These dirty operations are true realtime DBMS operations: they take the same predictable amount of time regardless of the size of the database.

3.7 Queries

Apart from traffic processing, telecommunications systems contain substantial amounts of operational maintenance (O & M) code. For example, when we delete a subscriber from a switching system, we need to search several tables for data which is associated with this subscriber, hence the need for a query language. Operational and maintenance code is characterized by the following properties:

1. It has none or very low realtime requirements.
2. It reads, searches and manipulates large parts of the traffic data.
3. It constitutes a large part of the code volume of the system.
4. It is seldom (if ever) executed, thus subject to software rot and consequently inherently buggy.

Thus, a powerful query language which executes on the target system and has complete access to all traffic tables, can remedy (3) by making the O & M code smaller and (4) by being declarative and by being able to automatically adapt to changes in table layout or network topology. Since an optimizing compiler is

used to decide the execution order of the query, O & M code can also become more efficient.

The Mnesia query language is based on list comprehensions. This idea has been exploited in several other functional DBMSs such as [15]. The syntax of list comprehensions blend perfectly with the Erlang programming language.

3.8 Schema Alteration

The Erlang programming language has extensive support to change the code of executing processes without stopping the process. It is possible to change the layout or organization of Erlang data without stopping the system. Thus, it is also possible to change the Mnesia schema at runtime without stopping the system. Since Mnesia is intended for nonstop applications, all system activities such as performing a backup, changing the schema, dumping tables to secondary storage and copying replicas have to be performed in the background while still allowing the applications to access and modify tables as usual. We believe that this is a requirement which is satisfied by few, if any, of the commercial DBMSs.

4 Some Implementation Aspects

Mnesia is entirely implemented in Erlang. The Erlang programming environment has turned out to be the ideal vehicle for the implementation of a distributed DBMS and the entire implementation of Mnesia including all aspects of the system from low level storage management to the optimizing query compiler is small and consists of not more than approximately 20000 lines of Erlang code.

Persistent storage is implemented on top of the underlying operating system file system. The disadvantage of this is performance of disc operations and the major advantage is portability. Since Mnesia is primarily intended to work as primary memory DBMS, we feel that the portability aspect is the more important. Tables and indexes in primary memory are implemented as linear hash lists [13] and secondary storage tables are implemented as named files. Each file is organized as a linear hash list with a medium chain length of the hash bucket set to a small value. The linear hash list is very efficient for lookup operations and reasonably efficient for insert operations. Files and tables can grow and shrink dynamically. Space management on each file is performed through a buddy algorithm.

The Mnesia lock manager uses a multitude of traditional techniques. Locking is dynamic, and each lock is acquired by a transaction when needed. Regular twophase locking [6] is used and deadlock prevention is traditional waitdie [14]. The time stamps for the waitdie algorithm are acquired by Lamport clocks [12] maintained by the transaction manager on each node. When a transaction is restarted, its Lamport clock is maintained, thus making Mnesia live lock free as well. The lock manager also implements multi granularity locking [10]. Traditional twophase commit [9] is used by the transaction manager when a transaction is commits.

Simple queries are evaluated by the relational DBMS technique operators [8], whereas recursive queries are evaluated by SLG [3] resolution. Since Mnesia is running on top of distributed Erlang the implementation is greatly simplified. In a distributed application there are separate Erlang nodes running on (usually) different machines. Erlang takes care of the communication between processes possibly on separate nodes transparently. Distributed Erlang works also transparently across different computers with different endianism, thus a Mnesia system can consist of a set of heterogenous computer systems. Processes and nodes can easily be started, supervised and stopped by processes on other nodes. This makes much of communication implementation difficulties disappear for Mnesia as well as for applications.

5 Performance Discussion

In this section we provide some measurements on the Mnesia system. The figures clearly indicate that:

- The cost of using the transaction system, as opposed to using the dirty interface, is substantial. We believe that the correct interpretation of this is that the dirty interface is fast and not that the transaction system is slow.
- The cost of replication is fairly high. This is expected since the computers used in the test are interconnected by an ordinary shared media 10 Mbit/sec Ethernet.

The computers in the test are three Sun UltraSparcs running Solaris 2.5. All transactions are initiated from the one UltraSparc at 167 Mhz and the other two machines run at 143 Mhz.

number of replicas	1	2	3
<code>divorce/1</code>	1877	5009	13372
<code>divorce/1</code> using <code>wread</code>	1225	4703	12185
dirty <code>divorce/1</code>	181	592	1121

Table 1. Wallclock microseconds to execute `divorce/1` with different configurations

In the first row, we run the function `divorce/1` from section 2. In the second row we run a version of `divorce` using a Mnesia function `wread/1` instead of `read/1`. This function reads the object but sets a write lock instead of a read lock. This is more efficient if we know that we are going to subsequently write the same object. This way the lock need not be upgraded from a read lock to a write lock. Finally in the last row we use the dirty functions to read and write the replicated tables, thus bypassing the transaction system and using the lightweight locks.

6 Conclusions

There exists today a very large number of DBMSs, a large number of commercially available systems as well as an uncountable number of research systems. It could appear as if it would be a better solution to use a commercial DBMS but if all the aspects from section 3 are taken in account, no suitable commercial DBMS exists. We feel that our main contributions are the following.

- We have implemented an entire distributed DBMS by combining a large number of well known techniques. Many research groups chose to study some aspects of DBMSs only, we have implemented a full distributed DBMS. Few such implementations exist.
- We have showed that Erlang is very well suited for not only telecommunications system but also for the implementation of DBMS systems like Mnesia. To our knowledge this is the first time a distributed DBMS is implemented in a symbolic programming language.
- We have provided a DBMS solution that address all, or at least many, aspects of data management in telecommunications systems.

The Mnesia system is currently being used to build real products in Ericsson today, thus it is no longer a mere prototype system, it has matured enough to be labeled a product. The system is available at <http://www.ericsson.se/erlang>

References

1. Armstrong, J. L., Williams, M. C., Wikström, C. and Viriding, S. R., Concurrent Programming in Erlang, 2:nd ed. Prentice Hall (1995)
2. Bernstein, P.A., Hadzilacos, V., Goodman, N. Concurrency Control and recovery in Database Systems Addison Wesley, 1987.
3. Chen, A.W., Warren, D.S. Query Evaluation under the WellFounded Semantics Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, Washington, 1993.
4. Case, K. McCloghrie, M. Rose, S. Waldbusser. Management Information Base for Version 2 of the Simple Network Management Protocol (SNMPv2), Jan, 1996.
5. Copeland, G., Maier, D. Making Smalltalk a database system Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data. pp. 316325. Boston 1984.
6. Eswaran, K.P., Grey, J.N., Lorie, R.A. and Traiger, I.L. The Notions of Consistency and Predicate Locks in a Database system Communications of ACM, 19(11):624633, November 1976.
7. Faehndrich, M., Morrisett, G., Nettles, S., Wing, J. Extensions to Standard ML to Support Transactions ACM SIGPLAN Workshop on ML and its Applications, June 2021, 1992.
8. Goetz, G. Query Evaluation Techniques for Large Databases ACMCS 2(25):73170, June 1993.
9. Grey, J.N. Notes on Database operating system: An advanced course Lecture notes in Computer Science, Springer Verlag, Berlin. 1(60):393481, 1978.

10. Grey, J.N., Lorie, R.A., Putzolo, G.R. and Traiger, I.L. Granularity of Locks and degrees of consistency in a shared database IBM, Research report RJ1654, September 1975.
11. Hvasshovd, SO., Torbjornsen, O., Bratsberg, S.E., Holager, P. The ClustRa telecom database: High availability, high throughput, and realtime response Proceedings of the 21st International Conference on Very Large Databases, Zurich, Switzerland, pp. 469477, September 1995.
12. Lamport, L. Time, clocks and the ordering of events i a distributed system ACM Transactions on Programming Languages and Systems, 21(1):558565, July 1978.
13. Larsson, PÅ Larsson. Dynamic Hash tables Communications of the ACM, 31(4), 1988
14. Rosenkrantz, D.J., Stearns, R.E. and Lewis, P.M. System Level Concurrency Control for Distributed Databases ACM Transactions on Database Systems, 3(2):178198, June 1978.
15. Trinder, P.W. and Wadler, P. List comprehensions and the Relational Cal culus Proceedings of the Glasgow 1988 Workshop on functional programming, Rothesay , August 1988, pp 115123.
16. Ullman, J. Principles of Database and KnowledgeBase Systems, vol 2. Computer Science press, 1989.

An AQUA-Based Intermediate Language for Evaluating an Active Deductive Object-Oriented Language *

Babu Siddabathuni, Suzanne W. Dietrich, and Susan D. Urban

Department of Computer Science and Engineering
Arizona State University, Tempe, AZ 85287-5406
{s.dietrich|s.urban}@asu.edu

Abstract. This paper presents an approach for evaluating the Comprehensive Declarative Object Language (CDOL). CDOL is a declarative language that supports the definition, retrieval and manipulation of data over an object-oriented database with support for active rules. We have designed and implemented an intermediate language for the evaluation of CDOL, which is based on the widely used *AQUA* object algebra. The mapping of a comprehensive application in CDOL to AQUA motivated this AQUA-based intermediate language, known as ABIL. A subset of AQUA operations that were necessary for translating CDOL to AQUA forms the basis of this intermediate language. This paper describes the design of ABIL and illustrates by example the translation of CDOL into ABIL. This paper also includes detailed examples of the evaluation of CDOL's rule-based query language, which utilizes a *binding structure* to maintain the bindings for the variables during the evaluation of a CDOL rule.

1 Introduction

The ADOOD RANCH [Diet92] project at Arizona State University is involved in the development of an environment that supports the analysis, testing and debugging of active rules in a deductive and object-oriented framework. The Comprehensive Declarative Object Language (CDOL) [Urba97] provides for the declarative definition, retrieval and manipulation of data over an object-oriented database with support for active rules. The rule-based query language is central to CDOL and provides the basis for the subcomponents of CDOL (constraint sublanguage, update sublanguage and active rule sublanguage). The constraint sublanguage provides for the declarative specification of integrity constraints to enforce database consistency. The update sublanguage provides the basis for the definition of update methods and transactions to manipulate data. The active rule sublanguage defines active rules that react to events of interest.

* This research is supported by NSF Grant No. IRI-9410993.

According to the characterization of the strategies for designing Deductive Object-Oriented Databases (DOODs) [Samp97], the design of the CDOL language follows a language reconstruction approach. The language reconstruction approach reconstructs a language (from scratch) that is designed to be a declarative language with object-oriented features, requiring the specification of an underlying theoretical framework. A formal semantics for CDOL has been specified [Kara96] using an order-sorted algebra. Other approaches to designing DOOD languages include language extension and language integration. The language extension approach extends an existing declarative, deductive language with object-oriented features, such as Coral++ [Sriv93]. The language integration approach integrates a declarative, deductive language with an imperative programming language in the context of an object model, such as VALIDITY [Frie95, Diet98]. [Samp97] also includes a comparison of DOOD systems on the support that they provide for various declarative and object-oriented features. However, it does not include any details on the evaluation of the DOOD languages.

This paper describes our approach to the evaluation of CDOL in the context of the ADOOD RANCH project and its component-based architecture [BenA97]. This ADOOD architecture requires fine grain support over objects and their deltas [Sund98] to support monitoring of events and conditions to allow for the analysis, testing and debugging of active rules.

Our approach to the implementation of the CDOL evaluation component was strongly influenced by the requirements of the ADOOD architecture and the overall goals of the research, focusing on the analysis, testing and debugging of active rules. Since the research was not emphasizing the *efficient* evaluation of CDOL, we did not set out to invent a WAM [AitK91] or a magic sets based evaluation strategy [Ullm89]. Our background literature survey uncovered a paper describing alternatives for evaluating a rule language over complex objects [Hene89]. These alternatives suggested the translation of the language to a relational algebra, a nested algebra or an object algebra. Based on the requirements of the ADOOD architecture and CDOL, the object algebra approach was chosen. Specifically, we chose the AQUA object algebra [Leim93].

AQUA has been designed to act as an intermediate language for a broad range of object-oriented query languages. An interpreter built for AQUA can evaluate any database language that can be translated into AQUA. However, our search for an AQUA interpreter was unsuccessful. We did not want to write an AQUA interpreter because it was not a major objective of our research goals. We did, however, translate CDOL to AQUA to gain an insight into the set of intermediate operations that an evaluator for CDOL needs to implement. We have designed and implemented an intermediate language for the evaluation of CDOL, which is based on a subset of the AQUA operators.

This paper describes the AQUA-based intermediate language, called ABIL, for the evaluation of CDOL. Section 2 presents an overview of CDOL and the evaluator component in the context of the ADOOD architecture. Section 3 motivates the mapping algorithm of CDOL into AQUA and provides an illustrative

example. Section 4 describes the design of ABIL and illustrates by example the translation of CDOL into ABIL. This section also includes detailed examples of the evaluation of CDOL’s rule-based query language, which utilizes a *binding structure* to maintain the bindings for the variables during the evaluation of a CDOL rule. Section 5 concludes the paper with a summary and discussion of future work.

2 CDOL and ADOOD Architecture

This section gives an overview of CDOL in the context of a horse racing database example [Urb97]. An overview of the different components of the ADOOD architecture and the interfaces that the CDOL evaluator has with these components is also illustrated.

CDOL uses the object definition language (ODL) of ODMG-93 [Catt95] for the foundation of its data definition language (DDL). CDOL extends the DDL to support the definition of virtual classes, virtual attributes, active rules and integrity constraints. The behavior of a class is declared through method signatures in the DDL and the implementation of these methods are defined declaratively in CDOL using update rules. These update rules form the basis of the update language, which is used for specifying methods, transactions and the actions of active rules. In addition to the update methods specified by the user, the system generates a set of default update methods. These system-generated methods support object creation, deletion and update operations on single-valued properties and insert/delete operations on set-valued properties. CDOL also supports the specification of transactions, which are a sequence of independent update rules not associated with a specific class.

Table 1 provides examples of CDOL rules for defining virtual classes and virtual attributes. Note that in CDOL, class and property names are identi-

<pre>rule expensive_rule { expensive_horses:Exp_Horse <- horses:Exp_Horse, Exp_Horse[price = The_Price], The_Price > 1000000; };</pre>	<pre>rule trainer_salary_rule { horses:Horse[salary_of_trainer = Salary] <- horses:Horse, Horse.trainer.salary = Salary; };</pre>
--	--

Table 1. Definitions of passive rules *expensive_rule* and *trainer_salary_rule*

fiers starting with a lowercase letter, and variables are identifiers starting with an uppercase letter. The passive rule *expensive_rule* defines the simple virtual class *expensive_horses* by deriving all the *horses* that have a *price* greater than

\$1,000,000. The passive rule *trainer_salary_rule* defines the bindings for the *horses* virtual attribute *salary_of_trainer*.

The active rule sublanguage provides for the definition of event-condition-action (ECA) active rules that monitor events of interest and trigger a predetermined response on the satisfaction of these conditions. CDOL also provides for the definition of condition-action (CA) active rules that trigger responses on the satisfaction of the conditions being monitored. Consider the following example of an ECA rule that gets a new *trainer* for a *horse* that is valued over \$250,000 if the salary of the trainer is less than \$75,000.

```
active trainer_salary
{
  event      after horses:Horse[update_price(New_Price)]
  condition  deferred New_Price > 250000,
              Horse.salary_of_trainer < 75000
  action     immediate horses:Horse[update_trainer(New_Trainer)]
              <- New_Trainer = get_trainer_over_75k();
}
```

This example illustrates several features of CDOL. The event *update_price* is a system-generated method for updating the *price* attribute of a horse. After the price of a horse is updated, the condition evaluation is deferred to the end of the transaction that updated the price. If the price of the *Horse* is greater than \$250,000 and the salary of the *Horse*'s trainer is less than \$75,000, then the action is immediately evaluated. The action consists of a single update rule that calls the system-generated *update_trainer* method with the bindings of the variable *New_Trainer*, which is derived by the evaluation of the body of the update rule. The *get_trainer_over_75k()* transaction executes an external function to identify an appropriate trainer.

The constraint sublanguage, which is based on the rule-based query language, provides for the specification of conditions that need to be satisfied by the database instance. Consider the following example of a constraint specifying that there must be 10 races in a race schedule.

```
constraint number_of_races
  if race_schedules:Schedule[races = Race]
  then count(Race) = 10;
```

The *races* property is a multi-valued attribute containing a collection of race objects. The aggregate operator *count* counts the number of races in the *Race* collection. The aggregate operators provided by CDOL (*count*, *sum*, *avg*, *product*, *min*, *max*) aggregate either on multi-valued properties of objects or single-valued properties of a set of objects.

The rule-based query language also provides for universal and existential quantification in the expression of the declarative rules. The universal quantifier *for_all* and the existential quantifier *exists* are defined over multi-valued properties of objects. A complete overview of the CDOL language is beyond the

scope of this paper. For the interested reader, the detailed grammar of CDOL is available at http://www.eas.asu.edu/~adood/CDOL_BNF/cdol_bnf.html.

An architecture [BenA97] that supports the testing, analysis and debugging of active rules over a deductive and object-oriented environment has been developed as part of the ADOOD RANCH project. Figure 1 defines the interfaces of the CDOL evaluator with the other components of the ADOOD architecture. The user specifies the applications and queries through the graphical user interface. The CDOL compiler is composed of several compilers-one for each of the sublanguages of CDOL. The CDOL compiler compiles an application, storing the results of compiling the DDL into the metadata. In addition, the CDOL compiler translates transactions and methods into the intermediate language and passes it to the metadata. Each of the three sublanguages of CDOL (update sublanguage, active sublanguage and constraint sublanguage) has a separate rule manager. Since updates to the database occur inside transactions, the update rule manager forms a subcomponent of the transaction manager. The condition monitor that monitors the conditions of CA rules forms the subcomponent of the active rule manager. The constraint manager manages the different constraints defined in the database. The object storage stores all the instances of the objects in the database. The metadata stores all the meta information about the class definitions and rules in the database. The CDOL language evaluator accepts the named metadata component and relevant bindings as input and evaluates it in coordination with the object storage and the metadata. If the named metadata component is an update rule, the evaluator will update the properties of the objects in the object storage. If the named metadata component is a condition, the evaluator will evaluate the condition and return the instances of the objects that satisfied the condition. The derived rule manager, which is a subcomponent of the language evaluator, evaluates the derived rules for the virtual classes and virtual attributes.

Although a formal semantics for CDOL has been specified [Kara96] using an order-sorted algebra, CDOL has been mapped to AQUA to provide an insight into the implementation of the CDOL language evaluator. An illustrative example mapping of a CDOL rule to AQUA along with the operators that formed the basis for the intermediate language are provided in the next section.

3 Overview of AQUA

AQUA [Leim93] is an object algebra that supports general object oriented concepts like encapsulation, collections of objects, inheritance, operations and notions of equality. It deals with basic types, such as integer and boolean, and abstract data types. AQUA supports different collections of objects, such as set, list, array, tree and graph and a rich set of operators on these collection types. AQUA was chosen as the object algebra in which to provide an additional theoretical meaning to CDOL since it has been designed to act as an intermediate language for a broad range of query languages. An interpreter built for AQUA can then evaluate any query language that can be mapped into AQUA. The

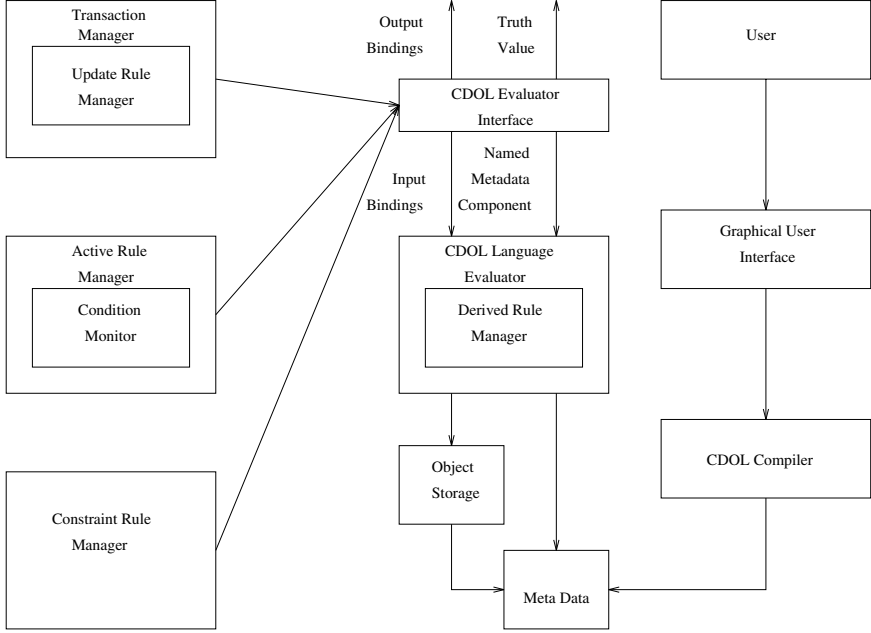


Fig. 1. Evaluator Architecture

subset of AQUA operators used in the mapping of a CDOL rule to AQUA is summarized in Table 2.

Without loss of generality, a normalized form of a CDOL rule has been defined to facilitate the mapping of CDOL to AQUA and the later mapping of CDOL to ABIL. In the following, let identifiers starting with an uppercase letter denote variables, such as V and W ; let identifiers starting with a lowercase letter, such as p and q , denote property names; and let c denote a constant. A literal of the form $V[p_1 = V_1, \dots, p_n = V_n]$, where p_i are the properties of the object referenced by variable V , is normalized to the sequence of literals $V[p_1 = V_1], \dots, V[p_n = V_n]$. A literal of the form $V[p \theta c]$ or $V.p \theta c$ where θ is one of the following $\{=, <, >, <=, >=\}$ is normalized to $V.p = W, W \theta c$ where W is a new variable. A valid path expression of the form $V.p.q$ must also be normalized. For example, the literal $V.p.q \theta c$ is normalized to $V.p = W, W.q = X, X \theta c$ where W and X are new variables.

The intuition for the mapping of CDOL to AQUA is based on the sideways information passing of the bindings in a left-to-right evaluation of the body of a CDOL rule. As each subgoal is evaluated, the AQUA expression maintains a tupling of the variable in the body of the rule with its values. Since we assume that the variables appearing in the head of the rule must be bound by the

<i>Operator</i>	<i>Definition</i>	<i>Description</i>
apply(f)(A)	$\{f(a) \mid a \in A\}$	returns the result set of elements by applying function f to each element in set A.
select(p)(A)	$\{a \mid a \in A, p(a)\}$	selects the set of elements from set A that satisfy predicate p.
mem(eq, a)(A)	$\exists x \in A. eq(a, x)$	returns true if there exists an element x in set A that is equal to input element a.
exists(p)(A)	$\exists a \in A. p(a)$	returns true if there exists an element a in set A that satisfies predicate p.
forall(p)(A)	$\forall a \in A. p(a)$	returns true if all the elements in set A satisfy predicate p.
join(p,f)(A,B)	$\{f(a, b) \mid a \in A, b \in B, p(a, b)\}$	adds an element f(a, b) to the result set if an element a from set A and an element b from set B satisfy predicate p(a, b).
tuple($l_1..l_n$)	$\langle l_1 : e_1, \dots, l_n : e_n \rangle$	takes a list of labels and data items and associates them pairwise.

Table 2. Selected Operators in AQUA

evaluation of the body of the rule, a projection on the distinguished variables in the head of the rule provides the result of the evaluation of the rule.

Consider the following mapping of the normalized CDOL expensive_horses rule to AQUA:

Passive rule for derived class expensive_horses (Normalized CDOL)

```

expensive_horses:Exp_Horse <-
    horses:Exp_Horse,                // S1
    Exp_Horse.price = The_Price,     // S2
    The_Price > 1000000;              // S3

```

Mapping to AQUA

```

Q1 = apply( $\lambda x$  tuple(Exp_Horse)(x))(horses)
Q2 = apply( $\lambda x$  tuple(Exp_Horse, The_Price)
    (x.Exp_Horse, x.Exp_Horse.price))(Q1)
Q3 = select( $\lambda x$  x.The_Price > 1000000) (Q2)
Query = apply( $\lambda x$  tuple(Exp_Horse)(x.Exp_Horse))(Q3)

```

The body of the rule has three subgoals S1, S2 and S3. The mapping of the first subgoal to AQUA produces a set of tuples Q1 of *horse* objects bound to the variable *Exp_Horse*. The mapping of the second subgoal produces a set of tuples Q2 of the *horse* objects along with their *price* bound to the variables *Exp_Horse* and *The_Price*, respectively. The mapping of the third subgoal Q3 involves the selection of the tuples from Q2 that have *The_Price* > \$1,000,000. Thus Q3 contains all the bindings for the variables in the body of the rule. The

evaluation of the head of the rule produces a set of tuples *Query* that gives the bindings for the virtual class *expensive_horses* by projecting on the variable *Exp_Horse*.

A comprehensive horse racing database application specified in CDOL has been mapped to AQUA. This mapping provided an intuition into the design of the intermediate operations and the data structure used in the evaluation of CDOL. A formal mapping algorithm of CDOL to AQUA is beyond the scope of this paper and is available in [Sidd98].

4 Intermediate Language

In this section, we illustrate by example the mapping of CDOL to the intermediate language ABIL and the data structure used in the evaluation of CDOL. CDOL is a declarative object language. The variables in the rules are bound to values during the evaluation of a rule. The evaluation of a CDOL rule proceeds from left to right. The bindings for the variables in a subgoal can be provided in three ways: parameters passed from the head of the rule, bindings provided by the underlying storage and bindings produced during the evaluation of previous subgoals through sideways information passing. These bindings are captured by the tuples in AQUA. This motivated the design of the *binding structure* that is responsible for capturing the values of these variables. In addition, the binding structure is also responsible for storing the type information of the variables. The binding structure evolves with the evaluation of the CDOL rule. It has a schema and a list of tuples. The schema is a list of fields and each field is composed of the name of the variable and its type information. The values of these variables are captured by a list of tuples. A schematic diagram of the binding structure is shown in Figure 2.

Table 3 shows the definition of *expensive_rule* in CDOL, normalized CDOL, AQUA and ABIL. The definitions have been provided in the various languages to give the reader an intuition on how the mapping of CDOL to AQUA provided the basis for the design of the intermediate language. The ABIL code is quite similar to the AQUA with extensions to include the typing information for the variables. The *apply_tuple* operation creates a new binding structure for the horse bindings associated with *Exp_Horse*. The *apply_member_tuple* operation adds the price member of *Exp_Horse* to the binding structure as *The_Price*. The *select* operation selects those horses having *The_Price* greater than \$1,000,000 and the variable *Exp_Horse* is projected as the result of the rule.

An example evaluation of the derived rule for the virtual class *expensive_horses* is shown in a 2-column format in Table 4. The left column shows the flow of evaluation of the intermediate operations and the resulting binding structure. The right column provides an explanation of the intermediate operation to its left in the context of this example.

The evaluation of any CDOL rule that has a virtual attribute or a virtual class involves the invocation of the Derived Rule Manager (DRM) by the language evaluator. This invocation may be recursive if the evaluation of the derived rule

requires the materialization of another derived rule. We now present an example of a passive rule *expensive_champ_rule* that has a virtual class *expensive_horses* in the body of the rule. This rule also illustrates the unnesting operation on a multivalued property.

Name ₁	Name ₂	Schema
Type ₁	Type ₂	
Value ₁ ¹	Value ₂ ¹	Tuples
Value ₁ ²	Value ₂ ²	
.....	
Value ₁ ⁿ	Value ₂ ⁿ	

Fig. 2. Binding Structure

Table 5 provides the definitions of the passive rule *expensive_champ_rule* in CDOL, normalized CDOL, AQUA and ABIL. The `apply_virtual_class_tuple` operation invokes the ABIL code for the *expensive_horses* virtual class, returning the resulting bindings for *E_C_Horse* in a new binding structure. The `apply_member_unnest_tuple` operation unnests the *race_record* member of *E_C_Horse* into *Race_Rec*, adding the appropriate bindings to the binding structure. The `apply_member_tuple` operation adds the *finish_position* member of *Race_Rec* to the binding structure as *Fin_Pos*. The select operation selects the bindings for a first place finish and the variable *E_C_Horse* is projected as the result of the rule.

Table 6 shows an example evaluation of *expensive_champ_rule* in a 2-column format, with the left column showing the resulting binding structure at each step and the right column providing a brief explanation.

The goal of the examples included in this paper is to give the reader an overview of some of the operations in the intermediate language. The list of operations illustrated in this paper is not an exhaustive one. Additional operations necessary for the evaluation of rules that involve negation, aggregation, existential and universal quantification are given in [Sidd98].

The CDOL evaluator and the derived rule manager subcomponent of the ADOOD architecture have been fully implemented in C++. The CDOL methods and transactions are translated to the intermediate language and stored in the metadata. These intermediate operations are then evaluated in conjunction with the object storage. Due to time constraints, the automatic translator that translates CDOL rules into ABIL has not yet been implemented. At this time,

<pre>expensive_horses:Exp_Horse <- horses:Exp_Horse, Exp_Horse[price = The_Price], The_Price > 1000000;</pre> <p style="text-align: center;">CDOL</p>
<pre>expensive_horses:Exp_Horse <- horses:Exp_Horse, Exp_Horse.price = The_Price, The_Price > 1000000;</pre> <p style="text-align: center;">Normalized CDOL</p>
<pre>Q1 = apply(λ x tuple(Exp_Horse)(x))(horses) Q2 = apply(λ x tuple(Exp_Horse, The_Price)(x.Exp_Horse, x.Exp_Horse.price))(Q1) Q3 = select(λ x x.The_Price > 1000000) (Q2) Query = apply(λ x tuple(Exp_Horse)(x.Exp_Horse))(Q3)</pre> <p style="text-align: center;">AQUA</p>
<pre>BS1 = apply_tuple(Exp_Horse, horses) apply_member_tuple(BS1, The_Price, int, Exp_Horse, price) select(BS1, The_Price, >, 1000000) project(BS1, [Exp_Horse])</pre> <p style="text-align: center;">ABIL</p>

Table 3. Definitions of the passive rule *expensive_rule*

the intermediate operations for the rules have been hand coded and stored in the metadata. A mapping algorithm that translates CDOL into the intermediate language has been defined and included in [\[Sidd98\]](#).

5 Summary and Future Work

This paper introduced the ABIL intermediate language, which is based on the widely used AQUA object algebra. The goal of ABIL is to provide the framework for the evaluation of CDOL, which is a declarative language that supports the definition, retrieval and manipulation of data over an object-oriented database with support for active rules. The design of ABIL was motivated by example mappings of CDOL into the AQUA object algebra. Some of the intermediate operations of ABIL were illustrated by detailed examples, including pictorial diagrams of the binding structure that maintains the bindings for the variables during the evaluation of a CDOL rule.

<table><tr><td>Exp_Horse</td></tr><tr><td>horses</td></tr><tr><td>h_1</td></tr><tr><td>h_2</td></tr><tr><td>h_3</td></tr><tr><td>h_4</td></tr><tr><td>h_5</td></tr><tr><td>h_6</td></tr></table> BS1 = apply_tuple(Exp_Horse, horses)	Exp_Horse	horses	h_1	h_2	h_3	h_4	h_5	h_6	Create a new binding structure <i>BS1</i> . The variable <i>Exp_Horse</i> and its type <i>horses</i> form the schema of <i>BS1</i> . All the instances of <i>horses</i> in the database provide bindings to <i>Exp_Horse</i> . h_1 , h_2 , h_3 , h_4 , h_5 and h_6 are the object identifiers of all the horses.								
Exp_Horse																	
horses																	
h_1																	
h_2																	
h_3																	
h_4																	
h_5																	
h_6																	
<table><tr><td>Exp_Horse</td><td>The_Price</td></tr><tr><td>horses</td><td>int</td></tr><tr><td>h_1</td><td>2000000</td></tr><tr><td>h_2</td><td>200000</td></tr><tr><td>h_3</td><td>500000</td></tr><tr><td>h_4</td><td>750000</td></tr><tr><td>h_5</td><td>1200000</td></tr><tr><td>h_6</td><td>1500000</td></tr></table> apply_member_tuple (BS1, The_Price, int, Exp_Horse, price)	Exp_Horse	The_Price	horses	int	h_1	2000000	h_2	200000	h_3	500000	h_4	750000	h_5	1200000	h_6	1500000	Append <i>The_Price</i> and its type <i>int</i> to the schema of <i>BS1</i> . For each tuple in <i>BS1</i> { get the value of the member attribute <i>price</i> of <i>Exp_Horse</i> ; bind this value to <i>The_Price</i> ; }
Exp_Horse	The_Price																
horses	int																
h_1	2000000																
h_2	200000																
h_3	500000																
h_4	750000																
h_5	1200000																
h_6	1500000																
<table><tr><td>Exp_Horse</td><td>The_Price</td></tr><tr><td>horses</td><td>int</td></tr><tr><td>h_1</td><td>2000000</td></tr><tr><td>h_5</td><td>1200000</td></tr><tr><td>h_6</td><td>1500000</td></tr></table> select(BS1, The_Price, >, 1000000)	Exp_Horse	The_Price	horses	int	h_1	2000000	h_5	1200000	h_6	1500000	Select the tuples from <i>BS1</i> that have <i>The_Price</i> > 1000000.						
Exp_Horse	The_Price																
horses	int																
h_1	2000000																
h_5	1200000																
h_6	1500000																
<table><tr><td>Exp_Horse</td></tr><tr><td>horses</td></tr><tr><td>h_1</td></tr><tr><td>h_5</td></tr><tr><td>h_6</td></tr></table> project(BS1, [Exp_Horse])	Exp_Horse	horses	h_1	h_5	h_6	Project the tuples in <i>BS1</i> on <i>Exp_Horse</i> .											
Exp_Horse																	
horses																	
h_1																	
h_5																	
h_6																	

Table 4. Evaluation of the passive rule *expensive_rule*

There are several open issues for future work. Although the CDOL evaluator and the derived rule manager subcomponent of the ADOOD architecture have been fully implemented in C++, several mapping components have not yet been implemented due to time constraints. These missing components include the mapping of CDOL to its normalized form and the mapping of normalized CDOL to ABIL. The definition of these mappings can be found in [Sidd98](#).

<pre>expensive_champ_horses:E_C_Horse <- expensive_horses:E_C_Horse, E_C_Horse[race_record = {Race_Rec}], Race_Rec.finish_position = 1;</pre> <p style="text-align: center;">CDOL</p>
<pre>expensive_champ_horses:E_C_Horse <- expensive_horses:E_C_Horse, E_C_Horse.race_record = {Race_Rec}, Race_Rec.finish_position = Fin_Pos, Fin_Pos = 1;</pre> <p style="text-align: center;">Normalized CDOL</p>
<pre>Q1 = apply(λ x tuple(E_C_Horse)(x))(expensive_horses) Q2 = apply(λ x tuple(E_C_Horse, Race_Rec) (x.E_C_Horse, x.E_C_Horse.race_record)(Q1) Q3 = unnest(Race_Rec)(Q2) Q4 = apply(λ x tuple(E_C_Horse, Race_Rec, Fin_Pos) (x.E_C_Horse, x.Race_Rec, x.Race_Rec.Fin_Pos))(Q3) Q5 = select(λ x x.Fin_Pos = 1)(Q4) Query = apply(λ x tuple(E_C_Horse)(x.E_C_Horse)(Q5)</pre> <p style="text-align: center;">AQUA</p>
<pre>BS1 = apply_virtual_class_tuple(E_C_Horse, expensive_horses, nil) apply_member_unnest_tuple(BS1, Race_Rec, entries, E_C_Horse, race_record) apply_member_tuple(BS1, Fin_Pos, int, Race_Rec, finish_position) select(BS1, Fin_Pos, =, 1) project(BS1, [E_C_Horse])</pre> <p style="text-align: center;">ABIL</p>

Table 5. Definitions of the passive rule *expensive_champ_rule*

Another open issue with respect to the CDOL evaluator component is its interaction with the condition monitor. A user can specify condition-action (CA) rules that monitor for conditions of interest and execute pre-specified actions on the satisfaction of these conditions. Object deltas [\[Sund98\]](#) that capture the incremental changes in the state of an object have been developed as part of the ADOOD RANCH project. The goal is to investigate the use of the deltas to provide bindings for the incremental evaluation of these conditions.

<table><tr><td>E_C_Horse</td></tr><tr><td>expensive_horses</td></tr><tr><td>h_1</td></tr><tr><td>h_5</td></tr><tr><td>h_6</td></tr></table> <div>BS1 = apply_virtual_class_tuple (E_C_Horse, expensive_horses, nil)</div>	E_C_Horse	expensive_horses	h_1	h_5	h_6	Create a new binding structure <i>BS1</i> . The variable <i>E_C_Horse</i> and its type <i>expensive_horses</i> form the schema of <i>BS1</i> . The oids h_1 , h_5 and h_6 of <i>expensive_horses</i> provide bindings to E_C_Horse. These oids have been derived by the evaluation of <i>expensive_rule</i> shown in Table 4. Since no parameters are used in the evaluation, a nil bind-structure is passed in.																
E_C_Horse																						
expensive_horses																						
h_1																						
h_5																						
h_6																						
<table><tr><td>E_C_Horse</td><td>Race_Rec</td></tr><tr><td>expensive_horses</td><td>entries</td></tr><tr><td>h_1</td><td>e_1</td></tr><tr><td>h_1</td><td>e_2</td></tr><tr><td>h_5</td><td>e_3</td></tr><tr><td>h_5</td><td>e_4</td></tr><tr><td>h_6</td><td>e_5</td></tr></table> <div>apply_member_unnest_tuple(BS1, Race_Rec, entries, E_C_Horse, race_record)</div>	E_C_Horse	Race_Rec	expensive_horses	entries	h_1	e_1	h_1	e_2	h_5	e_3	h_5	e_4	h_6	e_5	Append the variable <i>Race_Rec</i> and its type <i>entries</i> to the schema of <i>BS1</i> . The <i>race_record</i> member attribute of <i>E_C_Horse</i> is a set of <i>entries</i> . This set of <i>entries</i> are unnested and bound to <i>Race_Rec</i> .							
E_C_Horse	Race_Rec																					
expensive_horses	entries																					
h_1	e_1																					
h_1	e_2																					
h_5	e_3																					
h_5	e_4																					
h_6	e_5																					
<table><tr><td>E_C_Horse</td><td>Race_Rec</td><td>Fin_Pos</td></tr><tr><td>expensive_horses</td><td>entries</td><td>int</td></tr><tr><td>h_1</td><td>e_1</td><td>2</td></tr><tr><td>h_1</td><td>e_2</td><td>1</td></tr><tr><td>h_5</td><td>e_3</td><td>6</td></tr><tr><td>h_5</td><td>e_4</td><td>7</td></tr><tr><td>h_6</td><td>e_5</td><td>1</td></tr></table> <div>apply_member_tuple (BS1, Fin_Pos, Race_Rec, finish_position)</div>	E_C_Horse	Race_Rec	Fin_Pos	expensive_horses	entries	int	h_1	e_1	2	h_1	e_2	1	h_5	e_3	6	h_5	e_4	7	h_6	e_5	1	Append the variable <i>Fin_Pos</i> and its type <i>int</i> to the schema of <i>BS1</i> . For each tuple in <i>BS1</i> { get the value of member attribute <i>finish_position</i> of <i>Race_Rec</i> ; bind this value to <i>Fin_Pos</i> ; }
E_C_Horse	Race_Rec	Fin_Pos																				
expensive_horses	entries	int																				
h_1	e_1	2																				
h_1	e_2	1																				
h_5	e_3	6																				
h_5	e_4	7																				
h_6	e_5	1																				
<table><tr><td>E_C_Horse</td><td>Race_Rec</td><td>Fin_Pos</td></tr><tr><td>expensive_horses</td><td>entries</td><td>int</td></tr><tr><td>h_1</td><td>e_2</td><td>1</td></tr><tr><td>h_6</td><td>e_5</td><td>1</td></tr></table> <div>select(BS1, Fin_Pos, =, 1)</div>	E_C_Horse	Race_Rec	Fin_Pos	expensive_horses	entries	int	h_1	e_2	1	h_6	e_5	1	Select the tuples from <i>BS1</i> that have <i>Fin_Pos</i> = 1.									
E_C_Horse	Race_Rec	Fin_Pos																				
expensive_horses	entries	int																				
h_1	e_2	1																				
h_6	e_5	1																				
<table><tr><td>E_C_Horse</td></tr><tr><td>expensive_horses</td></tr><tr><td>h_1</td></tr><tr><td>h_6</td></tr></table> <div>project(BS1, [E_C_Horse])</div>	E_C_Horse	expensive_horses	h_1	h_6	Project the tuples in <i>BS1</i> on <i>E_C_Horse</i> .																	
E_C_Horse																						
expensive_horses																						
h_1																						
h_6																						

Table 6. Evaluation of the passive rule *expensive_champ_rule*

Acknowledgements

We would like to thank Taoufik Ben Abdellatif and Amy Sundermier of the ADOOD (Active, Deductive, Object-Oriented Databases) research group at Arizona State University for their comments on an earlier version of the paper.

References

- [AitK91] Ait-Kaci, H., "Warren's Abstract Machine," The MIT Press, 1991.
- [BenA97] Ben Abdellatif, T., "An Architecture for Active Database Systems Supporting Rule Analysis Through Evolving Database States," Ph. D. Dissertation Proposal, Arizona State University, August 1997.
- [Catt95] Cattell, R. G. G., "The Object-Oriented Database Standard: ODMG-93 Release 1. 2," Morgan Kaufmann Publishers, San Francisco, CA, November 1995.
- [Diet92] Dietrich, S. W., Urban, S. D., Harrison, J., and Karadimce, A. P., "A DOOD RANCH at ASU," in *Bulletin of the Technical Committee on Data Engineering* 1992, pp. 40-43.
- [Diet98] Dietrich, S. W., Friesen, O., and Calliss, F. W., "Rules and Objects in Database Systems: The VALIDITY Experience," *Submitted for publication*, 1998.
- [Frie95] Friesen, O., Gauthier-Villars, G., Lefebvre, A. and Vieille, L., "Applications of Deductive Object-Oriented Databases (DOOD) Using Datalog Extended Language (DEL)," in Ramakrishnan, R. ed., *Applications of Logic Databases*, Massachusetts: Kluwer Academic Publishers, 1995, pp. 1-22.
- [Heue89] Heuer, A., and Sander, P., "Semantics and Evaluation of Rules over Complex Objects," in *International Conference on Deductive and Object-Oriented Databases (DOOD)* 1989, pp. 473-492.
- [Kara96] Karadimce, A. P., "Termination and Confluence of Active Rules in Active Object Databases," Ph. D. Dissertation, Arizona State University, December 1996.
- [Leun93] Leung, T. W., Subramanian, B., Vandenberg, S. L., Mitchell, G., Vance, B., and Zdonik, S. B., "The AQUA Data Model and Algebra," in *Workshop on Database Programming Languages* 1993, pp. 157-175.
- [Samp97] Sampaio, P. R. F., and Paton, N. W., "Deductive Object-Oriented Database Systems: A Survey," in *Rules in Database Systems* 1997, pp. 1-19.
- [Sidd98] Siddabathuni, B., "Evaluation of the Comprehensive Declarative Object Language," M. S. Thesis, Arizona State University, December 1998.
- [Sriv93] Srivastava, D., Ramakrishnan, R., Seshadri, P., and Sudarshan, S., "Coral++: Adding Object-Orientation to a Logic Database Language," in *Proc. of the Intl. Conference on Very Large Databases* 1993, pp. 158-170.
- [Sund98] Sundermier, A., Ben Abdellatif, T., Dietrich, S. W., and Urban, S. D., "Object Deltas in an Active Database Development Environment," in *International Conference on Deductive and Object-Oriented Databases (DOOD)* 1997, pp. 211-228.

- [Ullm89] Ullman, J. D., "Principles of Database and Knowledge-Base Systems Volume 2," Computer Science Press, 1989.
- [Urba97] Urban, S. D., Karadimce, A. P., Dietrich, S. W., Ben Abdellatif, T., and Chan, H. W. R., "CDOL: A Comprehensive Declarative Object Language," in *Data & Knowledge Engineering* 1997, pp. 67-111.

Implementing a Declarative String Query Language with String Restructuring

Raul Hakli, Matti Nykänen, Hellis Tamm*, and Esko Ukkonen

Department of Computer Science

P.O. Box 26, FIN-00014 University of Helsinki, FINLAND

{Raul.Hakli,Matti.Nykanen,Hellis.Tamm,Esko.Ukkonen}@cs.Helsinki.Fi

Abstract. We describe the design and implementation of a declarative database query language for manipulating character strings. The language can be used to create logical predicates expressing structural properties of strings and relations between several strings. The predicates can be used to query strings in databases, and by leaving variables uninstantiated, also to generate new strings not contained in the database. A full working system was implemented as an extension of an object-oriented database management system and its query language. The declarative expressions are evaluated by first performing a compilation transforming them to nondeterministic finite state automata and then by simulating these automata using a depth-first search engine. The system checks the safety of each string-manipulation query in advance to preclude infinite ones. This safety checking provides also a compile-time loop-checking mechanism for the search engine, improving its efficiency.

1 Introduction

In the world of database systems and data management, the declarativeness of languages has traditionally been viewed as an ideal, perhaps more so than in programming language development and software engineering. One reason for the tendency may be that in database applications, where the users of the systems are often experts in other disciplines than computer science, the need for more intuitive ways of representing information is more urgent than in computer programming where the users of the languages are mostly computer professionals. Several widely used query languages, such as relation calculus, SQL, and Datalog [1], have their roots in the declarative paradigm, and of the new language extensions designed, the declarative ones are often preferred over the procedural ones because of their high-level nature and compatibility to the methods previously proved beneficial.

The need for query language extensions, in general, arises from some shortcomings of the original languages and data models, and also the currently used systems. One of them is the need for *string querying* and *string restructuring*, necessary for example in the field called bioinformatics.

* Supported by the Academy of Finland, grant number 22586.

Several methods have been proposed for string querying, for example Searls' String Variable Grammars [20], the PALM language [9], and the Proximal database query language extension [3]. Proposals for languages capable also of string restructuring, the ability to generate new strings not contained in the database, are fewer. The possibilities for string manipulation offered by Richardson's temporal logic are rather limited [19], but the transducer extensions of Datalog by Ginsburg & Wang [4] and Mecca & Bonner [13] seem powerful in theory. The working implementations capable of string restructuring based on the above theories are even fewer, however. According to our knowledge, only Mecca and Bonner have implemented such a prototype system.

In this paper, we continue our quest for a declarative language and its efficient implementation capable of expressing queries on strings as well as manipulating them [7,14,6,8,15,5]. The basis of our work is the *Alignment Calculus* of Grahne, Nykänen and Ukkonen, a modal logic designed for string querying. The theory of the calculus was laid in [7,14,8], the method for implementation was designed in [6,15], and the language itself, as well as the first prototype system without string restructuring capabilities, were introduced in [5]. Here, finally, we describe the first full implementation of the language we call the *Alignment Query Language*, in short *AQL*.

The main contribution of our work presented in this paper is the description of the first full implementation of Alignment Calculus [7] extending work presented in [5]. At the same time, it offers a detailed description of an implemented system providing a convenient string-oriented language suited for both string querying and restructuring. We also report some performance results presumably of interest to others working on string database implementations.

The paper is organized as follows: In Sect. 2 we describe AQL with some examples and give an overview of the implementation. In Sections 3 and 4 we study the central parts of our system in detail. The preliminary performance tests are discussed in Sect. 5 and the treatment ends with conclusions in Sect. 6.

2 Overall Architecture

In this section, we describe the architecture of our implementation. We start by sketching the AQL language giving some examples of its use. Then we give an overall description of the architecture of our system.

2.1 Language Description

The combination of an existing query language, in this case OQL [17], and our string handling extension based on Alignment Calculus, is what we call AQL. The string handling parts of AQL are called *Alignment Declarations* and they can be used to define predicates describing properties of individual strings or relations between several strings. These predicates can be used in OQL queries as selection conditions or to produce new strings in a given alphabet. For example, a two-place predicate `reverse(x, y)` can be used either to search the database

for pairs of strings that are reversals of each other or to produce reversed versions of the strings contained in the database. This resembles the question answering mechanism of Prolog language: If all the variables of a given predicate are instantiated, the evaluation returns a boolean value. Otherwise, the evaluator tries to find such values for the uninstantiated variables that the predicate will hold.

The Alignment Declarations are propositions about *alignments* of strings. An alignment consists of a set of strings and a *window column* through them:

```

      aat
      gc
gcaat

```

The alignments are connected to each other with accessibility relations called *transposes*. A transpose of an alignment is a one-step movement of one or several strings to the left or to the right. In alignments, it is possible to express propositions such as “the characters on the first and third rows positioned at the window column are not equal”. This proposition happens to be false in the alignment shown above, but it would be true in an alignment obtained from that alignment by a transpose sliding the third string one step to the left. Suppose now that the first, second, and third string are referred to using variable names x , y , and z , respectively. Using the Alignment Declarations, a proposition saying that after moving string z one step to the left, the window column characters on strings x and z are not equal, is expressed as `scan z on not x=z`. This proposition now holds for the alignment shown above.

Simple propositions like this one can be joined together to form more expressive ones. This can be done either by simple concatenation or by using more complex operators, repetition and choice. For example, a proposition `prefix(x,y)` saying that string x is a prefix of string y can be formulated as

```

repeat *times
  scan x,y on x=y
end
scan x on x=']'

```

The special character ‘`]`’, called the *right end marker*, denotes the end of a string, and the evaluation starts in an *initial alignment* where every string is located one step to the right of the window column, thus pointing to the *left end marker* ‘`[`’. Intuitively, this expression means that if all characters on strings x and y match until the end of string x , then string x is a prefix of string y . The `repeat`-loop can be abbreviated as `scan* x,y on x=y`, for it contains only one `scan`-clause.

Another example could be a two-place predicate which holds if either the first characters or the last characters of the input strings are equal:

```

choose
  scan x,y on x=y |
  scan* x
  scan* y
  scan x,y on x=y and not x=']'
  scan x,y on x=']' and y=']'
end

```

This predicate evaluates to true if either moving both strings one step forward results in equal characters on the window column, or if it is possible to proceed to such a position, that sliding one step forward on both results in matching characters not equal to the end marker and still one step further takes both strings to their ends. Note that the character ‘|’ separates the different choices that can be taken in the evaluation. There can also be more than two alternatives.

The basic elements of Alignment Declarations are introduced in Table 1. We use ϕ to denote *window formulae* which are boolean combinations of character comparisons, that is formulae of types $x=y$ or $x=a$ where x and y are string variables and a is a constant character. In addition, we have defined macros **scan*** and **rightscan*** for repeating the same **scan**- or **rightscan**-expression. Also there are macros like **read** and **rightread** for abbreviating multiple **scan**- or **rightscan**-expressions. For example, the expression **read x on "abc"** is an abbreviation for **scan x on x='a' scan x on x='b' scan x on x='c'**. Moreover, a structure like **repeat k times Φ end** will repeat the expressions in Φ for exactly k times and **repeat +times Φ end** will repeat the expressions in Φ at least once. A macro called **reset x** can be used to slide string x all the way back to the beginning.

Proposition	Explanation
skip	A tautology: always true.
scan x on ϕ	The proposition holds if ϕ holds after sliding string x one step to the left.
rightscan x on ϕ	The proposition holds if ϕ holds after sliding string x one step to the right.
$\Phi \Psi$	Iteration of modalities: the proposition holds if Φ holds in the initial alignment and if Ψ holds in the alignment obtained by making the transposes specified by Φ .
repeat *times Φ end	The proposition holds if the propositions in Φ repeated for zero, one, or several times hold.
choose Φ Ψ end	The proposition holds if either the propositions in Φ or the propositions in Ψ hold.

Table 1: The basic elements of Alignment Declarations

For being able to produce new strings, the alphabet used must be stated explicitly, which means that the variables have type declarations. Each predicate should begin by a **tape**-declaration stating the names of the variables and the alphabets used. For example, if the above two-place predicates were used for querying DNA sequences, they should begin with statement **tape x,y : DNA** where DNA is the name of the basic DNA alphabet consisting of the characters **a**, **c**, **g**, and **t**, denoting the different nucleic acids, adenosine, cytidine, guanosine, and thymidine, respectively. Different variables may have different types:

a predicate used for describing the transcription of DNA to RNA, for instance, could start with a statement **tape** x :DNA, y :RNA stating that the alphabet used for variable x is $\{a, c, g, t\}$ and the alphabet used for variable y is $\{a, c, g, u\}$.

The language also has a macro mechanism for re-using previously defined predicates. For example, let $\text{occurs}(x, y)$ be a predicate defined as

```
tape x,y:DNA
scan* y
scan x,y on x=y
```

Once defined and stored into the database, this predicate, true for the input strings denoted with x and y if string x is contained in string y , can be used inside another predicates definition in the following fashion:

```
tape x,y:DNA
repeat 3 times
  occurs(x,y)
reset x
end
```

This predicate holds if the DNA sequence x occurs in y at least three times. The **reset**-statement is used to rewind x back to the beginning while y keeps proceeding further.

2.2 System Architecture

In this subsection, we give a general overview of the system and discuss briefly the most central parts of the implementation. Fig. 1 shows a general diagram representing the operation of the system.

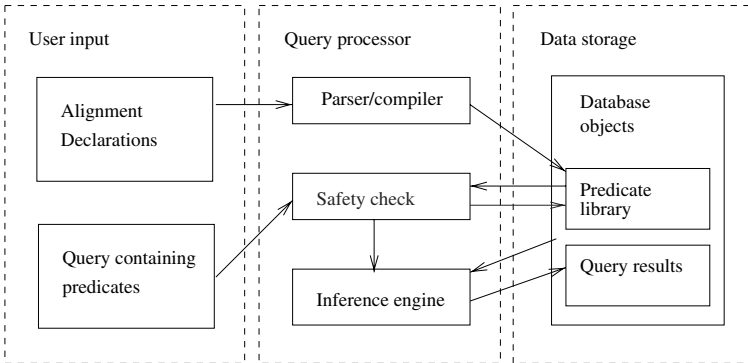


Fig. 1: The operation of the system

The user may create new predicates using the Alignment Declarations of AQL. The expressions are analyzed and a corresponding representation of a

finite state automaton is constructed and saved into the predicate library as a database object. When a query involving a string predicate is invoked by the user, a safety check must be performed. The result of the safety check depends on the combination of bound and free variables of the predicate. If the safety check fails, the query is discarded as infinite. Otherwise, the query is evaluated and the results, possibly containing newly generated strings, are stored into the database where they are available for use in subsequent queries.

The Database Management System. The system is implemented as an extension to the object-oriented database management system O₂ [11]. We needed a system allowing user-defined functions which are characteristic to object-oriented systems [2] and we wanted to test our ideas using a commercial database system, because our first prototype built on a freely distributed system had problems with efficiency and reliability [5]. O₂ was chosen because of its academic background and strong position in the object-oriented database market.

The Database Schema. With the requirements of bioinformatics in mind, we designed a database schema suitable for molecular sequence data, nucleotide sequences in particular. The database schema is an object-oriented version of the flat file data model used in the EMBL Nucleotide Sequence Database [23] containing fields like the molecule type, species, taxonomy, publication date, and other necessary information about each nucleotide sequence entry. Our main interest, however, is in the sequence field which is a character string of varying length in a restricted alphabet.

Predicate Creation. When the predicates are introduced using the Alignment Declarations, the expression is parsed recursively in a top-down manner and an intermediate representation is produced. The intermediate representations are transformed into nondeterministic two-way multi-tape automata and stored into the database. A detailed description is given in Sect. 3.

Safety Check. For each k -place predicate created, there are 2^k different combinations of free and bound variables. The user must specify which combination she wants to use when making a query involving a string predicate. The first time a new combination is introduced on a predicate, a safety check is made in order to find out if it is safe to use the predicate in the way specified by the user. For example, it is safe to use the predicate `prefix(x, y)` when x is free and y is bound because the answer set, that is, all prefixes of string y in a finite alphabet, remains finite when y is known. Using the predicate by specifying x and letting the system generate y is unsafe, however, for even in a finite alphabet there is an infinite number of strings which string x is a prefix of. This kind of a query must thus be inhibited. The result of the safety check is stored into the database for future reference. The safety check is discussed in detail in Sect. 3.2.

Simulation. For each k -ary predicate, a corresponding nondeterministic two-way k -tape acceptor is constructed and stored into the predicate library. The user can formulate a query involving a predicate with both free and bound variables. In the world of the automata, the bound variables correspond to tapes for which there is *input* to be placed and the free variables represent the *output* tapes for which new strings are to be generated. The idea, thus, is to use acceptors as transducers as explained in [15].

If all the tapes of the automaton are specified as input tapes, the simulation proceeds in a standard recursive fashion returning true if and only if the predicate given holds for the input strings. If there is output to be generated, the simulation will produce all the tuples of strings for which the predicate holds. How this is carried out is explained in Sect. 4.

3 Compilation of AQL

This section explains how the Alignment Declarations introduced in Sect. 2.1 are translated into a machine-executable form. For a declaration with k distinct variables, this form is a *k -tape nondeterministic two-way finite state acceptor with end-markers* (a *k -FSA* for brevity), with variables and tapes in one-to-one correspondence given by their order in the **tape** declaration. Fig. 2 provides as an example a 2-FSA, which accepts a pair of strings in alphabet $\{a, b\}$ exactly when the second is the reversal of the first; in other words, it implements the predicate **reverse**(x, y) alluded to in Sect. 2.1 and expressed as follows.

```
tape x,y:AB_alphabet
scan* x on not x=']'
scan x on x='['
repeat * times
  scan y on not y=']'
  rightscan x on x=y
end
scan y on y='['
rightscan x on x='['
```

A transition of the form

$$p \xrightarrow[d_1, \dots, d_k]{c_1, \dots, c_k} q \quad (1)$$

in a k -FSA \mathcal{A} means naturally “if in state p the reading head of each tape $1 \leq i \leq k$ is scanning the character c_i , then \mathcal{A} can as its next computation step enter state q , while moving each head i one square to the left ($d_i = +1$), or right ($d_i = -1$), or keeping it stationary on the current square ($d_i = \pm 0$)”. A detailed description can be found elsewhere [14, Section 3.1] [15, Section 1.1].

The translation procedure is explained in Sect. 3.1, while the safety analysis mentioned in Sect. 2.2 is explained in Sect. 3.2.

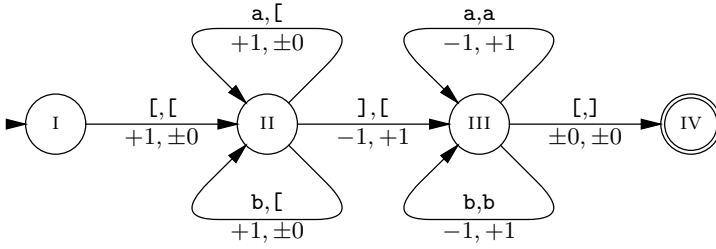


Fig. 2: A 2-FSA for recognizing string reversals.

3.1 Forming and Representing Automata

Let Φ be an Alignment Declaration with variables x_1, \dots, x_k on alphabets $\Sigma_1, \dots, \Sigma_k$. Noting that the basic elements of Alignment Declarations given in Table 1 specify a *regular expression* of (right)scan-propositions, the first step of the translation is to build a finite automaton [10, Chapter 2] corresponding to Φ , whose transitions are of the form

$$p \xrightarrow{(\text{right})\text{scan } \mathbf{x} \text{ on } \phi} q. \quad (2)$$

In fact, an earlier implementation without string restructuring [5] used such automata as its executable representation.

The next step is to convert each transition of the form (2) into ones of the form (1). This is attained by first adding a new intermediate state r , dividing the transition (2) into $p \xrightarrow{(\text{right})\text{scan } \mathbf{x}} r \xrightarrow{\text{on } \phi} q$, and then converting the first half into the appropriate tape movements, and the second half into testing the resulting character combination.

The first half simply generates a transition $p \xrightarrow[c_1, \dots, c_k]{d_1, \dots, d_k} r$ on every possible character combination c_1, \dots, c_k and determines the corresponding tape movements d_i according to \mathbf{x} and the scanning direction, taking into account the property of Alignment Declarations that trying to scan past an end-marker keeps the head stationary instead.

The second half generates then a stationary transition $p \xrightarrow[0, \dots, 0]{c_1, \dots, c_k} q$ for every character combination c_1, \dots, c_k , which satisfies the condition ϕ after replacing each variable x_i with character c_i . Intuitively, this group of transitions spells out the “truth table” of the formula ϕ .

The translation of a single scan-proposition given above is costly, because it expands both parts into $O(\prod_{i=1}^k (|\Sigma_i| + 2))$ transitions. However, this explicit expansion can be defended in three ways: (a) In our application, the alphabets tend to be fairly small; 4 for DNA and RNA, for example. (b) The number k tends also to be small; in fact, the current implementation requires $k \leq 5$. This is because in many cases Alignment Declarations with many variables can be equivalently expressed as combinations of simpler ones: for example “ x is a common substring of y and z ” divides into testing both “ $\text{occurs}(x, y)$ ” and “ $\text{occurs}(x, z)$ ”, where

occurs was introduced in Sect. 2.1. (Admittedly, not every string operation permits such a division [4, Theorem 4].) (c) The subsequent safety analysis of this Alignment Declaration becomes easier and more accurate, because the effects of its single scan-propositions are now spelled out explicitly.

Carrying case (c) further, the translator finally performs the following *flow analysis* on \mathcal{A}_Φ , the k -FSA produced. Consider two consecutive transitions

$$p \xrightarrow{c_1, \dots, c_k} r \xrightarrow{d'_1, \dots, d'_k} q, \quad (3)$$

where some $d_i = \pm 0$, but $c_i \neq c'_i$. We know that if r is entered via the first transition, it cannot be exited via the second. These incompatible pairs are eliminated from \mathcal{A}_Φ by expanding every state r into states $\langle r, e_1, \dots, e_k \rangle$, where each $e_i \in \Sigma \cup \{[,], \mathbf{L}, \mathbf{R}\}$ remembers in the finite control of \mathcal{A}_Φ , what happened on tape i when entering r : \mathbf{L} (\mathbf{R}) means it was scanned left (right), while the other symbols mean that it stayed on the given character. The formulation of this flow analysis was simplified by the explicit translation method for single scan-propositions given above; analyzing a more implicit translation would have entailed more work in this phase.

\mathcal{A}_Φ is now almost ready: it only remains to delete by standard means those states not on any path from the initial (I in Fig. 2) into the accepting state (IV in Fig. 2), and those stationary transitions that do not enter the accepting state.

The (transition graph of the) automaton \mathcal{A}_Φ is currently represented and stored as adjacency lists implemented with the persistent object-oriented data structures offered by the underlying O_2 system.

3.2 Analyzing the Safety of Automata

An Alignment Declaration Φ appears within a database query q , which can provide values for some, but not necessarily all, its free variables x_1, \dots, x_k . For example, Φ might say that “ x_1 is the concatenation of x_2 and x_3 ”, while the rest of q says that the values for x_1 come from a database table, but rely on Φ to split each value into a prefix x_2 and a suffix x_3 . On the one hand, we would like our system to operate as if Φ merely selects rows from an infinite database table containing the set $\Sigma_1^* \times \Sigma_2^* \times \Sigma_3^*$ of all string triples for the corresponding alphabets Σ_1 , Σ_2 and Σ_3 . On the other hand, the evaluation of q must remain finite.

A database view of the situation is that the string relation expressed by Φ must satisfy the *finiteness dependency* $\Gamma = \{x_1\} \rightsquigarrow \{x_2, x_3\}$ [18] (a subcase of *functional dependencies* [11, Chapter 8.2]). A programming view is that Γ specifies an *input-output annotation* for Φ , declaring x_1 as input, and x_2 and x_3 as output. Then we ask whether there are inputs capable of producing infinitely many outputs or not.

In this latter view, whether a given finiteness property Γ holds for a given Alignment Declaration Φ is solved in our system by examining the corresponding k -FSA \mathcal{A}_Φ built as in Sect. 3.1. The problem is known to be undecidable in

```

1: Determine all the maximal strongly connected components of  $G$ ;
2: Delete from  $G$  all transitions between different components;
3: for all components  $G'$  of  $G$  do do
4:   if  $G'$  has a clock: an input tape, which moves in some transitions, but always in
     the same direction then
5:     Delete from  $G'$  all transitions that move this chosen clock;
6:     if calling this algorithm recursively on  $G'$  yields false then
7:       return false
8:     end if
9:   else if  $G'$  still has transitions then
10:    return false
11:   end if
12: end for
13: return true

```

Algorithm 1: The recursive part of the safety analyzer.

general [14, Section 4.1] and decidable if at most one tape moves into both directions [14, Section 4.2]. However, this decision algorithm is infeasible in practice, and therefore our system uses instead a heuristic method [14, Algorithm 1] [15, Algorithm 1], which (a) never declares an unsafe annotation safe, (b) works even in some cases with more than one tape moving right as well as left, and (c) correctly determines the safety of all cases with all tapes moving left only.

Our method is related to a method for inferring the same annotations for logic programs [12], which proceeds by examining the strongly connected components in the call graph of a logic program. However, because Alignment Declarations are iterative rather than recursive, and because the flow analysis mentioned in Sect. 3.1 makes the interactions between variables explicit, our method remains simpler.

The method has two parts: **Part 1** checks that the transition graph G of \mathcal{A}_Φ contains no path from the start into the final state, where some output tape is never tested for ‘]’; intuitively, every accepting computation must “close all its output files” with the end-marker. **Part 2** consists of calling Algorithm 1 on G to verify that every loop consumes irrevocable “input clock ticks”, and therefore terminates at least when they run out. \mathcal{A}_Φ is deemed safe for a given annotation, if both parts yield **true**.

Let us see how this method performs on the 2-FSA \mathcal{A}_{rev} in Fig. 2, when tape 1 is declared input and tape 2 output. Part 1 succeeds because of the only transition into the final state IV. Part 2 begins by identifying each state as a separate component, and deleting all transitions between them. Component I is a single state without any transitions, so it neither recurses nor exits the loop with failure. Tape 1 suffices as a clock for component II, and this results in a recursive call to Algorithm 1 with just the single state II. This call returns with success, following the behaviour of component I. Component III behaves similarly to component II, and component IV similarly to component I. Hence part 2 returns with success, and thus \mathcal{A}_{rev} is safe for this annotation.

On the other hand, the method cannot verify the finiteness of \mathcal{A}_{rev} when tape 2 contains the input: part 2 fails for component II because tape 2 is stationary and cannot therefore act as clock, and thus this component cannot be reduced into a transition-free one. In other words, our method is “myopic” in the sense that it – very prudently – considers the loop in state II unsafe, because this loop guesses nondeterministically any possible output. Later looping in state III will in fact eventually reject all but finitely many of these tentative outputs, but our method does not look that far ahead.

The current implementation uses the method outlined above to determine whether the Alignment Declarations Φ within a query q do satisfy given input-output annotations. On the other hand, it does not select the annotations to try by looking at q ; instead, the user must tell the system the direction, in which each Alignment Declaration is to be used when forming an evaluation plan for q . A theory for selecting a query evaluation plan automatically in the presence of these annotations has been developed based on the theory of finiteness dependencies [6][14, Section 4.4]. However, implementing it would require access to the internals of the O_2 query planner, which has not been necessary for the other parts of our system. On the other hand, achieving good performance in databases with user-defined data types does seem to call for lowering the barrier between the query planner and the user-defined procedures [22].

4 Execution of AQL

Here we explain the algorithms our system uses for evaluating the k -FSA a given Alignment Declaration has been translated into. Sect. 4.1 considers the case, where all k tapes receive input, while Sect. 4.2 extends it to the case, where some of these tapes are declared as output instead. Finally, Sect. 4.3 explains how the generated outputs are processed.

4.1 The Depth-First Search Engine

Let us first consider the case, where a k -FSA \mathcal{A} is to be executed as an acceptor with all input strings w_1, \dots, w_k given to find out whether \mathcal{A} accepts them. Because \mathcal{A} is nondeterministic, this means finding some sequence of computation steps, which leads from the initial configuration (where the state of \mathcal{A} is the start state and each tape i contains $[w_i]$ with its head on the ‘[’) into an accepting configuration (where the state is the accepting one). Our solution is to use *depth-first search* as shown in Algorithm 2 from the initial configuration.

The major drawback is that \mathcal{A} might repeat a configuration within one branch of the (implicit) search tree, leading Algorithm 2 into an infinite loop, unless we use some kind of *loop checking* to determine whether the current configuration has occurred before. This seems undesirable when good performance is sought.

However, our system never proceeds with the execution without consulting the safety analyzer of Sect. 3.2 first. This seems unnecessary here, because no output is being generated. Far from it: a successful execution of Algorithm 1

```

1: if the current configuration  $C$  is accepting then
2:   return true
3: else
4:   for all transitions  $\tau$  applicable in  $C$  do
5:     if calling this algorithm recursively on the configuration that results from
       taking  $\tau$  in  $C$  yields true then
6:       return true
7:     end if
8:   end for
9: end if
10: return false

```

Algorithm 2: A depth-first acceptor simulator.

can be shown [14, Section 5.2.1] to imply that no configuration can ever repeat. Therefore our system uses Algorithm 2 with *compile-time* loop checking. (We have also experimented with a variant of Algorithm 2, which sought to avoid repeating configurations in different branches as well [11], but this proved to be intolerably slow because of all the necessary bookkeeping.)

We used the C programming language instead of the object-oriented extensions provided by O₂C [16] in this computationally intensive part of our system, because we found it quite difficult to restrict or even determine beforehand their resource consumption.

4.2 The Production of New Strings

Let us now turn to the case, where inputs are provided only for some tapes of a k -FSA \mathcal{A} , and the system is asked to produce all the possible contents of the other tapes as output. Put another way, \mathcal{A} should now act as a (generalized) *transducer* [10, Chapter 2.7], or automaton with output. Recall that our system assures that the current input-output annotation has indeed been deemed safe for \mathcal{A} by the method explained in Sect. 3.2.

Declaratively, the easiest way to extend FSAs into transducers is to declare the output tapes *write-once* by introducing a new blank character ‘ \sqcup ’, initializing each semi-infinite output tape with $\sqcup\sqcup\sqcup\dots$, and defining that matching any character c against ‘ \sqcup ’ succeeds when selecting the next transition, but results also in c replacing ‘ \sqcup ’ in the square under the tape head in question.

Algorithm 2 accommodates this, if we add an **else** branch to the **if** statement in lines 5–7, where each ‘ \sqcup ’ that was replaced when trying τ is subsequently restored. The result is an algorithm, which returns **true** when it finds the first output tuple, which can then be found on the tapes. Part 1 of the safety analysis method ensures that every output tape j is then of the form $[v_j]\sqcup\sqcup\sqcup\dots$, where $v_j \in \Sigma_j^*$ is the corresponding output string. This algorithm can be further modified to report every output tuple (in a way described in Sect. 4.3), if we change line 2 to report these strings v_j before returning **false**.

Does our modified algorithm terminate? Our safety analyzer from Sect. 3.2 in fact ignores the output tapes altogether. Therefore as in Sect. 4.1, successful

safety analysis ensures that looping cannot occur, and we have a straightforward depth-first computation strategy with compile-time loop checking even when output is being produced.

4.3 Eliminating Duplicate Answers

When used as a transducer as in Sect. 4.2, a given k -FSA \mathcal{A} can report the same answer tuple multiple times, because as a nondeterministic acceptor \mathcal{A} may well have multiple accepting computations on the same input. For example, the predicate `occurs(x, y)` mentioned in Sect. 2.1 reports every substring x of y for as many times as x occurs within y . Should they be considered as different answers, as in an object-oriented database model [1, Chapter 21], which might endow each with a separate identity, or just “echoes” of the one real answer, as in a relational database model [1, Chapter 3]?

We chose the latter, and thus our system must report each output only once. Therefore whenever the evaluator of Sect. 4.2 has an answer to report, it concatenates the current contents of the output tapes (including each `]`) into one string u , and stores this u into a *trie* data structure T [21, Chapter 17], unless already there. This T starts out empty, and collects in this way all the different u that the evaluator reports for the current inputs, thereby recognizing duplicates whenever they appear.

5 Performance Evaluation

We experimented with different automata using different combinations of bound and free variables. Some results are presented in Table 2. The tests were performed on Sun SPARCserver 670/MP on Unix platform with light system workload.

As input in our experiments we used substrings of suitable lengths of ten sequences from the EMBL Nucleotide Sequence Database [23]. So we performed ten series of simulations using those strings as input for every automaton, input/output tapes combination, and string length in consideration. The results presented indicate the average values of eight simulation results so that the minimum and maximum values of the ten results are not taken into account.

The meaning of the columns in the table is the following. The first column `STRLEN` contains the input string length. The second column `WALLTIME` reports the overall time (“wall clock time”) of a simulation including reading the input strings from the database, writing the output strings to the database and some necessary checks: if the automaton exists, if the safety check has been performed for the current input/output tapes combination, etc. The column `GENTIME` shows the user CPU time taken by the depth-first search engine, but without the duplicate elimination time. The fourth column `#TRANS` indicates the number of transitions taken by the search engine in the automaton graph and the next column gives us the number of transitions taken per second of `GENTIME`. The column `DUPLTIME` shows the user CPU time taken by duplicate elimination and

SIMPLE CONCATENATION							
STRLEN	WALLTIME	GENTIME	#TRANS.	#TRS./SEC.	DUPLTIME	#RESULTS	#CHARS
250	2.50	1.29	126508	98368.26	0.00	1	500
500	6.47	5.14	503008	98023.15	0.06	1	1000
750	12.78	11.45	1129508	98704.36	0.02	1	1500
1000	21.90	20.38	2006008	98427.49	0.00	1	2000

SMART CONCATENATION							
STRLEN	WALLTIME	GENTIME	#TRANS.	#TRS./SEC.	DUPLTIME	#RESULTS	#CHARS
250	1.19	0.01	1005	88317.53	0.00	1	500
500	1.23	0.02	2005	90561.97	0.00	1	1000
750	1.39	0.03	3005	91419.03	0.00	1	1500
1000	1.54	0.04	4005	92666.62	0.01	1	2000
1250	1.51	0.05	5005	92574.09	0.01	1	2500
1500	1.53	0.07	6005	92279.44	0.01	1	3000

SIMPLE SPLIT							
STRLEN	WALLTIME	GENTIME	#TRANS.	#TRS./SEC.	DUPLTIME	#RESULTS	#CHARS
250	3.77	0.60	64260	106647.88	0.15	251	62750
500	8.36	2.47	253510	102708.27	0.64	501	250500
750	15.31	5.43	567760	104640.80	1.45	751	563250
1000	26.03	9.62	1007010	104725.21	2.76	1001	1001000

SMART SPLIT							
STRLEN	WALLTIME	GENTIME	#TRANS.	#TRS./SEC.	DUPLTIME	#RESULTS	#CHARS
250	3.60	0.59	64005	109351.18	0.09	251	62750
500	8.72	2.40	253005	105587.13	0.35	501	250500
750	15.35	5.32	567005	106515.23	0.77	751	563250
1000	25.18	9.33	1006005	107854.70	1.50	1001	1001000

REVERSAL							
STRLEN	WALLTIME	GENTIME	#TRANS.	#TRS./SEC.	DUPLTIME	#RESULTS	#CHARS
250	1.28	0.02	2260	131633.86	0.00	1	250
500	1.23	0.03	4510	133989.90	0.00	1	500
750	1.25	0.05	6760	134609.30	0.00	1	750
1000	1.36	0.07	9010	134510.55	0.00	1	1000
1250	1.38	0.08	11260	134191.73	0.00	1	1250
1500	1.43	0.10	13510	133690.75	0.01	1	1500

SUBSTRING OCCURRENCE							
STRLEN	WALLTIME	GENTIME	#TRANS.	#TRS./SEC.	DUPLTIME	#RESULTS	#CHARS
50	5.69	0.03	4029	138960.19	0.03	1175	21914
100	20.25	0.12	15554	130744.52	0.17	4791	171147
150	56.05	0.24	34579	144008.27	0.54	10887	572804

Table 2: Results of performance measurements.

the last two columns report the number of outputs and characters in the output, respectively. All time measurements are given in seconds. (Timings close to zero are especially subject to fluctuations in the workload.)

The results presented in this paper are produced by simulations with `reverse` (from Sect. 3), `occurs` (from Sect. 2.1) and concatenation automata. We defined two concatenation predicates (simple and smart concatenation) and we ran the corresponding automata in two directions, first to concatenate two strings into a third, and then to split one string into its prefix-suffix pairs as in Sect. 3.2

The `simple_concatenation` predicate can be expressed as

```
tape x,y,z:DNA
scan* x,y on x=y
scan* x,z on x=z
scan x,y,z on x=']' and y=']' and z=']'
```

The `smart_concatenation` predicate can be written as

```
tape x,y,z:DNA
scan* x,y on x=y and not y=']'
scan y on y=']'
scan* x,z on x=z and not x=']'
scan x,z on z=']' and x=']'
```

Both of the predicates hold if string x is a concatenation of strings y and z . However, there is a great difference in performance between these two concatenations which is mainly caused by the third line of latter predicate. Namely, the smart concatenation first matches the whole string y to a prefix of x and after that it goes on to match z to the rest of x while the simple concatenation combines every prefix of y with all prefixes of z and tries to match such concatenations to x . So, the performance of the simple concatenation automaton is quadratic with respect to the input string lengths while the smart concatenation performs linearly. This can be well seen when comparing the `GENTIME` columns in the first and second subtables of Table 2. However, the next two subtables show that these two automata when run in split direction perform rather similarly.

Finding a reversal of a string behaves linearly as one would expect while generating all substrings of a given string takes asymptotically quadratic time with respect to the input string length (the last two subtables).

For rough comparison, we measured the speed of two Prolog-programs running on the same platform as our system. We used version 3.1.1 of SWI-Prolog [24] to reverse a list of characters and to produce all sublists of a given list of characters. Taking the averages as explained above, we obtained the following results: For lists of length 250, 500, 750, 1000, 1250, and 1500, the user CPU times in seconds to produce a reversed list were 0.00, 0.01, 0.01, 0.02, 0.02, and 0.02, respectively. For lists of length 50 and 100, the times to produce all substrings were 0.18 and 1.16, respectively.

While the linear problem of producing a reversed version of a list seems to be faster using Prolog, our system wins clearly in the substring production, which

is more complex. This gives us faith that our system may actually prove to be useful in nontrivial molecular sequence querying.

The substantial difference in performance between the two versions of concatenation predicate shows the sensitivity of a declarative language like ours to small modifications in the syntactic form of the expression. An interesting research problem would be to study if it is possible to convert predicates written using Alignment Declarations into logically equivalent but more efficiently evaluable forms.

These preliminary tests were made using rather short strings so it is still quite difficult to say how efficient our system would be in querying actual biological sequence databases. However, although the databases contain some very long sequences, most of the sequences are fairly short. For example, the average length of nucleotide sequences in the EMBL database is less than 700 characters [23]. It would be interesting to measure the performance of our system on complete queries with sequences of varying lengths but these kinds of tests still remain to be done.

6 Conclusions

This paper has presented an implementation of AQL, a declarative language for string querying and restructuring based on modal Alignment Calculus [7]. The system was implemented as an extension to O₂ database management system and it can be used in string-oriented application areas such as bioinformatics where the nucleotide sequences are represented as character strings. The system operates by first parsing and compiling the predicates written in AQL to produce finite-state automata and then simulating them in a depth-first fashion. The finiteness of the queries is ensured by safety analysis and the simulation is speeded up by performing compile-time instead of run-time loop-checking. The back-end of the system being now completed, we will shift emphasis to user-interface design in the future.

References

1. Abiteboul, S., Hull, R., and Vianu, V. *Foundations of Databases: the Logical Level*. Addison-Wesley 1995.
2. Atkinson, M., Bancilhon, F., DeWitt, D. et al. The object-oriented database system manifesto. In *Deductive and object-oriented databases: Proceedings of the First International Conference on Deductive and Object-Oriented Databases (DOOD89)* (1989), pp. 223–240.
3. Ganguly, S., and Noordewier, M. Proximal: a database system for the efficient retrieval of genetic information. *Computers in Biology and Medicine* 26, 3 (1996), 199–207.
4. Ginsburg, A., and Wang, X.S. Regular sequence operations and their use in database queries. *Journal of Computer and System Sciences* 56, 1 (1998), pp. 1–26.

5. Grahne, G., Hakli, R., Nykänen, M., and Ukkonen, E. AQL: an alignment based language for querying string databases. To appear in *Ninth International Conference on Management of Data (COMAD'98)*.
6. Grahne, G., Nykänen, M. Safety, translation and evaluation of Alignment Calculus. In *Proceedings of the First East-European Symposium on Advances in Databases and Information Systems (ADBIS'97)* (1997), pp. 295–304.
7. Grahne, G., Nykänen, M., and Ukkonen, E. Reasoning about strings in databases. In *Proceedings of the 13th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (1994), pp. 303–312.
8. Grahne, G., Nykänen, M., and Ukkonen, E. Reasoning about strings in databases. To appear in the *Journal of Computer and System Sciences (JCSS)*.
9. Helgesen, C., and Sibbald, P.R. PALM - a pattern language for molecular biology. In *Proceedings of the First International Conference on Intelligent Systems in Molecular Biology* (1993), pp. 172–180.
10. Hopcroft, J.E., and Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley 1979.
11. Kekkonen, M., Koskelainen, J., Niemi, J., Tuononen, T., Vihervaara, A., and Vuolasto, J. Design document for the BiO₂ system (in Finnish). Tech. rep., Department of Computer Science, University of Helsinki, Finland, 1997.
12. Krishnamurthy, R., Ramakrishnan, R., and Shmueli, O. A framework for testing safety and effective computability. *Journal of Computer and System Sciences* **52** (1996), pp. 100–124.
13. Mecca, G., and Bonner, A.J. Sequences, Datalog and transducers. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (1995), pp. 23–35.
14. Nykänen, M. *Querying String Databases with Modal Logic*, PhD thesis, Department of Computer Science, University of Helsinki, Helsinki, 1997.
15. Nykänen, M. Using acceptors as transducers. To appear in *Third International Workshop on Implementing Automata (WIA'98)*.
16. *O₂C User Manual*. O₂ Technology 1996.
17. *OQL User Manual*. O₂ Technology 1996.
18. Ramakrishnan, R., Bancilhon, F., and Silberschatz, A. Safety of recursive Horn clauses with infinite relations (extended abstract). In *Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (1987), pp. 328–339.
19. Richardson, J. Supporting lists in a data model (a timely approach). In *Very Large Data Bases Conference* (1992), pp. 127–138.
20. Searls, D.B. String variable grammar: a logic grammar formalism for the biological language of DNA. *The Journal of Logic Programming* **24**, 1 & 2 (1995), 73–102.
21. Sedgewick, R. *Algorithms*, 2nd edition. Addison-Wesley 1988.
22. Seshadri, P., Livny, M., and Ramakrishnan, R. The case for extended Abstract Data Types. In *Very Large Data Bases Conference* (1997), pp. 66–75.
23. Stoesser, G., Sterk, P., Tuli, M.A., et al. The EMBL Nucleotide Sequence Database. *Nucleic Acids Research* **25**, 1 (1997), pp. 7–13.
24. Wielemaker, J. *SWI-Prolog Reference Manual*. University of Amsterdam, The Netherlands, <http://www.swi.psy.uva.nl/usr/jan/SWI-Prolog/Manual/Title.html>, 1997.

Client-Side Web Scripting with HaskellScript

Erik Meijer¹, Daan Leijen¹, and James Hook²

¹ Utrecht University
Department of Computer Science
{erik,daan}@cs.uu.nl

² Oregon Graduate Institute
Pacific Software Research Center
hook@cse.ogi.edu

Abstract. Using client-side scripting it is possible to build interactive web pages that don't need round-trips to the server for every user-event. The web browser exposes itself to the script via an object model (DOM), which means that scripts can add and remove page content, or change the position and color of elements via their style attributes. We explain the object model as implemented by Microsoft Internet Explorer by means of examples and report on our experiences of using Haskell as a programming language for client-side web scripting using the *HaskellScript* scripting engine.

1 Introduction

Since the introduction of the web there has been a constant migration of computational power from the server to the client. Server-side scripting using CGI [12] provides critical interactive data entry capabilities and dynamically generated content, but in a very explicit, modal, turn-taking style of interaction. Client-side scripting extends the interaction paradigm by allowing content designers to specify reactive behaviors in their web pages in a much more modular and efficient fashion. For example, with client-side web scripting it is possible to locally make the appearance of an HTML element change when the mouse is positioned over the element, without the need of generating a completely new page on the server.

The reactive model exported by the browser to the scripting language interpreter is essentially a mechanism for registering call-back functions on events as well as access functions for the rich object structure of the browser and its contained HTML document. The use of a lazy, higher-order, typed language like Haskell [3] as a scripting language matches the event/call-back model very well because it supports the necessary low-level imperative features, but it also allows encapsulation and hiding of these gory details into powerful, reusable high-level abstractions.

The official W3C HTML 4.0 specification [14] allows for embedded scripts in a document, but it does not provide a standard object model that describes how

scripts can manipulate the document content. Unfortunately, the Netscape and Microsoft browsers have incompatible DOM versions. The recently announced DOM Level 1 specification [4] defines a standard API, but it is not yet fully implemented in the standard browsers. In this paper we will therefore concentrate on the object model as implemented by Microsoft's Internet Explorer 4 [6], which in contrast to Navigator makes *all* objects in a document scriptable, and provides an elegant, language independent, open framework for plugging in non-standard scripting languages (section 5). We are currently working on an implementation of the full DOM Level 1 API on top of the IE5 object model.

The remainder of this paper is organized as follows. Section (2) gives a short introduction to Haskell, in particular to the IO-monad and the `do{}` notation and gives references for further reading. Section 3 develops a motivating example, the classic nervous text applet in Haskell. Section 4 first explains how COM Automation objects are invoked from Haskell; it then continues by exploring the different objects that make up the DOM. The underlying ActiveX-Scripting architecture is the subject of section 5. The paper concludes in section 6.

2 Minuscle Introduction to Haskell

When interacting with the outside world or accessing object models, we have to deal with side-effects. In Haskell [1], effectful computations live in the IO monad [7, 9, 16]. A value of type `IO a` is a *latently* effectful computation that, *when executed*, will produce a value of type `a`. The command `getChar :: IO Char` will read a character from the standard input when it is executed.

Like any other type, latently effectful computations are first class citizens that can be passed as arguments, stored in list, and returned as results. For example `putChar :: Char -> IO ()` is a function that takes a character and then returns a computation that, when executed, will print that character on the standard output.

Effectful computations can be composed using the `do{}`-notation. The compound command `do{ x <- mx; mf x } :: IO b` is a latent computation, that, when executed, first executes the command `mx :: IO a` to obtain a value `x :: a`, passes that to the action-producing function `mf :: a -> IO b` and executes `(mf x)` to obtain a value of type `b`. For example `do{ c <- getChar; putChar c }` is a command that, when executed, reads a character from the standard input and copies it to the standard output.

In other scripting languages such as JavaScript, commands are not first class values and are represented as code *strings*. The JavaScript `eval` method takes a code string argument, which is parsed and then executed. This is obviously a very primitive and error-prone way of composing effectful behaviour. It wrecks all hope for a static type system, and, even worse, it loses static scoping since the code passed to the `eval` method is executed in the same context as the call to the `eval` method.

In this paper we adopt style conventions that emphasize when we are dealing with effectful computations. Specifically, all expressions of type `IO` are written with an explicit `do{}`. In the same vein, values of functional type `a -> b` are written as lambda-expressions `\x -> e`. To reflect the influence of the OO style, we will use postfix function application `object # method = method object` to mimic the `object.method` notation. These conventions result in highly stylized programs from which it is easy to tell the type of an expression by looking at its syntactic shape.

3 Nervous Text

A classic web page animation is the eye catching “nervous text” applet, which randomly jitters the characters of a sentence.

Ha^s kell i sCool

To script this in Haskell, we define a HTML page with a `<DIV>` container¹ to hold the animated sentence. The container has an `ID="text"` attribute, so that we can refer to it in the script via the call `window # getItem "text"`. The element `<SCRIPT SRC="Nervous.hs" LANGUAGE="HaskellScript">` instructs the browser to load the `HaskellScript` engine and execute the `main` function of module `Nervous.hs`:

```
<STYLE>.a {position:relative}</STYLE>
<DIV ID="text" >Haskell is Cool</DIV>
<SCRIPT LANGUAGE="HaskellScript" SRC="Nervous.hs"></SCRIPT>
```

The `main` function of module `Nervous.hs` splits the content of `DIV` element into the individual letters and installs an event handler that bounces these around at every `interval` milliseconds (library functions like `getWindow`, `item`, and `setInterval` are explained in section 4).

```
main :: IO ()
main =
  do{ window <- getWindow
      ; division <- window # getItem "text"
      ; letters <- division # splitToLetters
      ; window # setInterval (do{letters # bounce}) interval
    }
```

¹ `DIV` abbreviates “division”. A document division is essentially a labeled block in HTML.

Function `splitLetters` wraps every character of its argument inner text into a `SPAN` element², inserts this as HTML, and finally returns the newly produced list of children:

```
splitToLetters :: IHTMLElement -> IO [IHTMLElement]
splitToLetters = \element ->
  do{ cs <- element # getInnerText
    ; element # setInnerHTML
      (unlines ["<SPAN CLASS=\"a\">"+[c]+"</SPAN>" | c <- cs])
    ; element # getChildren
  }
```

In conjunction with the `STYLE` element, the `CLASS="a"` attribute makes the position of all animated elements relative to their normally rendered position. The Haskell prelude function `unlines :: [String] -> String` turns a list of strings into a single string separated by newlines.

An `IHTMLElement` object can be moved around on a page by changing the `PixelLeft` and `PixelTop` properties of its `style` attribute:

```
getStyle :: IHTMLElement -> IO IStyle
setPixelLeft, setPixelTop :: Int -> IStyle -> IO ()
```

Function `move` combines these three functions into a compound method that is more convenient to use:

```
move :: (Int,Int) -> IHTMLElement -> IO ()
move = \ (newLeft,newTop) -> \element ->
  do{ style <- element # getStyle
    ; style # setPixelLeft newLeft
    ; style # setPixelTop newTop
  }
```

Assuming a function `random :: IO Float` that returns a random number between 0.0 and 1.0, we can randomly reposition an element using `moveRandom`:

```
moveRandom :: IHTMLElement -> IO ()
moveRandom = \element ->
  do{ newLeft <- random; newTop <- random
    ; element # move (scale jump newLeft,scale jump newTop)
  }
```

Function `scale jump :: Float -> Int` scales the random number to an integer in the range `-jump to jump :: Int`.

Finally, function `bounce` maps `moveRandom` over a list of `IHTMLElements` to create the groovy effect:

² The `SPAN` tag defines the scope of style attributes. In this case, every character has an independent set of style attributes.

```

bounce :: [IHTML_Element] -> IO ()
bounce = \elements ->
  forEach_ elements $ \element -> do{element # moveRandom}

```

The function `forEach_ :: [a] -> (a -> IO ()) -> IO ()` applies a monadic, side-effecting function to each element in a list. It can easily be defined in terms of the standard monadic map function `mapM_`.

That's all. A complete rocking and rolling web-page in just a few lines of plain Haskell.

4 The HTML Object Model

The HTML object model is an API through which all document content of a web page, including elements and attributes, is programmatically accessible and manipulable. This means that scripts can add, remove and change elements and/or tags and navigate through document structure at will. The visual rendering of HTML objects (such as font properties, colors, backgrounds, and box properties) is controlled by style sheets [10], which are fully scriptable as well. HTML elements are exposed to the Haskell programmer as COM Automation objects [8, 2] of type `IDispatch` `d`.

To enjoy a good cup of Java, you don't have to be an expert on growing beans. Similarly, to use `HaskellScript`, no knowledge of COM is required. Just a basic understanding of monadic IO is sufficient. Automation objects are special COM components that implement the `IDispatch` interface [13]. An example Automation interface is `IHTMLLocation` whose IDL interface description looks like:

```

[uuid(3050F25E-98B5-11CF-BB82-00AA00BDCE0B)]
interface IHTMLStyle : IDispatch {
    ...;
    [propput] HRESULT color([in] VARIANT p);
    [propget] HRESULT color([out, retval] VARIANT* p);
    ...;
    HRESULT toString([out, retval] BSTR* String);
    ...;
}

```

The `color` property is of type `VARIANT` to allow different color representations like string color names such as `"red"` and hexadecimal RGB encodings such as `#FF0000`.

Arguments and results types of properties and methods of Automation interfaces are limited to a fixed set of `VARIANT` types. The type class `Variant` contains two member functions, the first one `marshal :: Variant a => a -> Pointer VARIANT -> IO ()` marshals a Haskell value into a `VARIANT` and the second

`unmarshal :: Pointer VARIANT -> IO a` unmarshals a `VARIANT` into its corresponding Haskell representation:

```
class Variant a where
  { marshal :: a -> Pointer VARIANT -> IO ()
  ; unmarshal :: Pointer VARIANT -> IO a
  }
```

The real `Variant` class has some additional member functions as it uses a trick due to Phil Wadler [15] to simulate overlapping instances akin to the `Read` and `Show` classes in Haskell.

We have written an IDL compiler called `H/Direct` [5] that generates Haskell bindings for COM components whose interfaces are specified in IDL. The compiler maps the properties and methods of the `IHTMLStyle` interface into corresponding Haskell functions that take an interface pointer to the `IStyle` as their last argument:

```
type IStyle = IDispatch ()

getColor :: (Variant a) => IStyle -> IO a
getColor = \c -> \style -> do{ style # get "color" }

setColor :: (Variant a) => a -> IStyle -> IO ()
setColor = \c -> \style -> do{ style # set "color" c }

toString :: IStyle -> IO String
toString = \style -> do{ selection # method_0_1 "toString" }
```

4.1 Properties

The (overloaded) functions `get` and `set` inspect the properties of an Automation object. They take the name of the property as a string argument:

```
set :: (Variant a) => String -> a -> IDispatch d -> IO ()
get :: (Variant a) => String -> IDispatch d -> IO a
```

4.2 Methods

The overloaded family of functions `method_n_m` is used to call the methods of Automation objects from Haskell. Function `method_n_m` maps an n `Variant` arguments to an m -tuple of `Variant` results:

$$\begin{aligned} \text{method_n_m} :: & \overbrace{(\dots, \text{Variant } a, \dots)}^{n>0}, \overbrace{(\dots, \text{Variant } b, \dots)}^m \\ & \Rightarrow \text{String} \rightarrow \dots \rightarrow a \rightarrow \dots \rightarrow \text{IDispatch } d \\ & \rightarrow \text{IO } (\dots, b, \dots) \end{aligned}$$

Internally, `get`, `set`, and `method_n_m` are implemented using `marshal` and `unmarshal` to marshal and unmarshal arguments to and from `VARIANTS` and low level functions to call raw COM components. Note that `marshal value` is a function that when given a pointer, marshal `value` in the memory to which that pointer points:

```
set member = \value -> \obj ->
do{ dispid <- obj # getMemberID member
  ; p <- obj # primInvokeMethod dispPROPERTYSET False dispid
    [marshal value] []
  ; freeMemory p
}
```

4.3 Events

All HTML elements can generate and handle events such as user events (e.g. mouse clicks), update events, and change events. Events bubble upwards from the source element to the root of the object tree and can be caught along the way. Handling events requires that the client of an Automation object (in this case the script written by the user) implements outgoing interfaces (event sinks, or callbacks) that will be called by the event source object when the event occurs. The `HaskellScript` library function `onEvent_n_m` installs an event sink for a given component:

$$\begin{aligned} \text{onEvent_n_m} &:: \overbrace{(\dots, \text{Variant } a, \dots)}^{n>0}, \overbrace{(\dots, \text{Variant } b, \dots)}^m \\ &\Rightarrow \text{String} \rightarrow (\dots \rightarrow a \rightarrow \dots \rightarrow \text{IO } (\dots, b, \dots)) \\ &\rightarrow \text{IDispatch} \rightarrow \text{IO } () \end{aligned}$$

In order to accommodate for scripting languages without first class functions, all HTML events have zero arguments and return no results. Instead, they use a global event object that contains information about the most recent event.

4.4 The IWindow Object

The root of the HTML object model is the `IWindow` object, which represents the entire browser. Scripts can call `getWindow :: IO IWindow` to obtain a handle to their window object.

The `IWindow` object is quite rich, it has about 40 properties, 30 methods and can handle 10 different events. Amongst these are methods for standard GUI widgets such as `alert` that pops up an alert box, `confirm` that asks for confirmation, and `prompt` that ask the user to enter a string:

```
alert    :: String -> IWindow -> IO ()
confirm  :: String -> IWindow -> IO Bool
prompt   :: String -> String -> IWindow -> IO String
```

From `IWindow`, we can descend down the object tree using the following access functions

```
getHistory    :: IWindow -> IO IHistory
getNavigator  :: IWindow -> IO INavigator
getLocation   :: IWindow -> IO ILocation
getScreen     :: IWindow -> IO IScreen
getEvent      :: IWindow -> IO IEvent
getDocument   :: IWindow -> IO IDocument
```

The `IHistory` object contains the list of URLs that the browser has visited; it is the programmatic interface to the “forward” and “backward” buttons in the IE toolbar.

The `INavigator` object contains information about the browser itself. JavaScript programs often use the `INavigator` object to distinguish between Internet Explorer and Netscape Navigator.

The `ILocation` object is the programmatic interface of the “address bar” of the browser. The various properties of the `ILocation` object dissect the currently loaded URL into its constituents (`hash`, `host`, `hostname`, `href`, `pathname`, `port`, `protocol`, and `search`). The three `ILocation` object methods `assign`, `reload`, and `replace` control the currently displayed document.

The `IScreen` object contains various properties of the physical screen, the most important of these are `width` and `height`.

The `IEvent` object is a ‘global variable’ that contains the parameters of all events as they occur. For instance, the `srcElement` property gives the `IHTMLElement` on which the last event occurred and there are various other properties like `screenX` and `screenY` from which the (relative) location of a button-click can be computed.

4.5 Example: Scrolling Status Bar

The `IWindow` object has a `status` property which contains the text that is currently displayed in the status bar of the window. Many webpages have *scrolling* status bars to attract attention. The idea behind a scrolling statusbar is to pad the status text on the left with spaces, and then remove them one after the other at every clock tick.

```
msg = take 50 (repeat ' ') ++ "..."
```

```
main =
  do{ window # setInterval (do{window # scroll}) scrollInterval}
```

```
scroll :: IWindow -> IO ()
scroll = \window ->
```

```
do{ msg' <- window # getStatus
  ; window # setStatus (if msg==" " then msg else (tail msg'))
}
```

4.6 The IDocument Object

Much of the dynamism of dynamic HTML comes from the access to the window's `IDocument` object. Through this object we can access the individual `IHTMLElement` objects that make up the page, but also the document's background color, and much much more. For example, the following script will change the background color to red when the user clicks anywhere on the document:

```
main :: IO ()
main =
  do{ window # getWindow
    ; document <- window # getDocument
    ; document # onClick (do{document # setBgColor "red"})
  }
```

The function `getItem` that returns a handle to a named element is in fact an abbreviation for a compound method that finds the element in the document's `all` collection:

```
Window.getItem :: String -> IWindow -> IO IHTMLElement
Window.getItem = \name -> \window ->
  do{ window # (getDocument ## getAll ## Document.getItem name)}
```

4.7 The IHTMLElement and IStyle Objects

As we have seen in the nervous text example, we can change properties of the `style` property of `IHTMLElements` to create interesting effects. Amongst the 75 properties of the `IStyle` object are properties like

- `fontSize`, `fontStyle`, `fontVariant`, and `fontWeight` to change the font using which the parent `IHTMLElement` is rendered,
- `pixelLeft`, `pixelTop` and `zIndex` that control the element's position, and
- `visibility` and `display` that control the element's visibility.

Clicking on the button displayed by the following HTML, will cause the text within the `SPAN` element to disappear and reappear:

```
<BUTTON ID="toggle">Toggle</BUTTON>
<SPAN ID="text">Click the button to make me (dis)appear</SPAN>
```

The underlying script installs an event handler for the `onclick` event that will alternately set the `display` property of the `style` property of item `"text"`:

```

main :: IO ()
main =
  do{ window <- getWindow
    ; text <- window # getItem "text"
    ; button <- window # getItem "toggle"
    ; button # onClick (do{text # toggle})
  }

toggle :: IHTMLElement -> IO ()
toggle = \element ->
  do{ style <- element # getStyle
    ; display <- style # getDisplay
    ; style # setDisplay (if display=="" then "none" else "")
  }

```

The textual representation of an `IHTMLElement` may be queried and modified through the four properties `innerHTML` and `outerHTML`, and `innerText` and `outerText`. The `innerHTML` property represents element without the enclosing tag of the element itself. Similarly the `innerText` property represents the contained HTML without any markup at all. The `outerHTML` and `outerText` properties are similar to `innerHTML` and `innerText` respectively except that they include the element's begin- and end-tags too. This makes it possible to remove or replace complete elements from a document completely.

As an example of using the `innerHTML` property, we will write a script to rotate between a list of HTML elements within a `SPAN` element [6](#):

```

Create
<SPAN ID="word" words="["<EM>amazing</EM>","<B>funky</B>"]">
</SPAN>
webpages with HaskellScript!

```

The list of words is stored inside a custom attribute `words :: [String]`. For the programmer, there is no distinction between custom and built-in attributes:

```

setWords :: [String] -> IHTMLElement -> IO ()
setWords = \words -> \element ->
  do{ element # setAttribute "words" (show words) }

getWords :: IHTMLElement -> IO [String]
getWords = \element ->
  do{ words <- element # getAttribute "words"; readIO words }
  'catch' (\_ -> do{ return [""] })

```

The call `readIO words` raises an exception if `words` cannot be parsed as a list of strings. If an exception occurs, the exception is caught and `getWords` returns a singleton list containing the empty string.

The `main` function for this example just installs an interval timer to rotate through the word list every `interval` milliseconds:

```
main :: IO ()
main =
  do{ window <- getWindow
    ; word <- window # getItem "word"
    ; window # setInterval (do{word # rotate}) rotateInterval
  }

rotate :: IHTMLElement -> IO ()
rotate = \element ->
  do{ (word:words) <- element # getWords
    ; element # setInnerHTML word
    ; element # setWords (words++[word])
  }
```

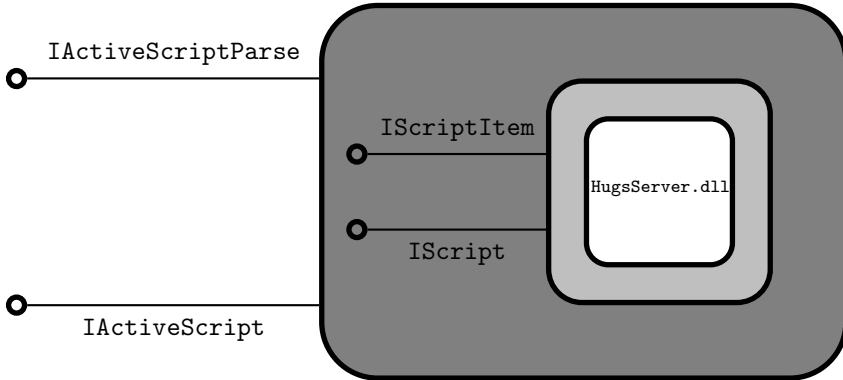
5 ActiveX Scripting, the Enabling Technology

The browser needs to communicate with the scripting language's execution engine in order to execute the scripts it encounters in the web page. The ActiveX Scripting architecture [2] is a language-independent protocol that enables embedded scripts in a variety of scripting host applications, including the Internet Explorer browser, the IIS web server and Windows itself. If a scripting host wants to execute a script, it will look if there is scripting engine available for the particular language in which the script is written. If so, it will create an instance of the engine, send it the script code, make its root objects available, and run the script. For the browser the root object is the `IWindow` object.

The ActiveX scripting framework is language- and application-independent. Any application that implements the `IActiveScriptHost` interface can be scripted in any language for which the `IActiveScript` interface is available. Traditionally most scriptable applications such as CAD-tools, editors, and workflow-tools have hardwired, application specific, internal scripting languages. This means that we cannot leverage our knowledge of say vi-macros when we use Emacs or \TeX . ActiveX scripting makes it possible to use a single scripting language for a wide range of applications, or many scripting languages within a single application.

To implement an ActiveX scripting engine, we have to create a COM object that supports the `IActiveScript` and `IActiveScriptParse` interfaces. The `IActiveScript` interface provides the methods necessary to initialize the scripting engine. One of its most important methods is `AddNamedItem`, which is called by the host to inject objects into the engine's namespace, and `Close`, which causes the scripting engine to shut down. The `IActiveScriptParse` interface provides the ability to load scripts and evaluate expressions, primary via the `ParseScriptText` method.

Instead directly implementing complete versions of the `IActiveScript` and `IActiveScriptParse` interfaces on top of the Yale/Nottingham Hugs interpreter, we have made a generic script engine that uses a much simpler set of interfaces than required by the full ActiveX Scripting framework. Our `IScript` and `IScriptItem` interfaces represent the minimal functionality needed for a language in order to be used as a scripting engine.



The benefit of this layering is that all the C++ adapter code that implements the Active Scripting interfaces, using the script server interfaces, can be reused to implement a scripting engine for any other language (for example Standard ML), or Haskell interpreters other than Hugs. This is considerably simpler than implementing the ActiveX scripting engine interfaces from scratch.

An additional advantage is that the script server interfaces are easy to use from ordinary, non scripting host, applications. This makes it possible have a very close interoperation between languages like Visual Basic or Delphi with good graphical development environments, or applications like Microsoft Office or Visio that have hardwired scripting languages but that can use Automation objects. The Hawk group at OGI has taken this route to implement a Visio-based GUI for their functional hardware description language [11].

The IDL specification for our script server interfaces is:

```
interface IScript : IDispatch {
    HRESULT AddHostItem([in]BSTR itemName, [in]IDispatch* item);
    HRESULT Load([in]VARIANT* scriptSource);
    HRESULT GetScriptItem
        ([in]BSTR name, [out,retval]IScriptItem** item );
};

interface IScriptItem : IDispatch {
    HRESULT Eval( [in] int cArgs,
                  , [in,size_is(cArgs)] VARIANT* args,
                  , [out,retval] VARIANT* result
                );
};
```

A host loads a script by calling `Load`. The script's source can be specified with a file name but also with an `IStorage` object, which is used by the ActiveX scripting engine to load a script that resides in memory. When a module is loaded, its `main` function is evaluated if present. The `AddHostItem` method injects host objects into the script's context. The browser uses it (indirectly) to make the `window` object available, and a VB program could use it to make a button or form visible to the script. In the script, objects added by `AddItem` can be retrieved using function `getHostItem :: String -> IO (IDispatch a)`.

In the other direction, hosts can use `GetScriptItem` to get access to variables, objects and functions that exported from the script. To the host these Haskell values are exposed via the `IScriptItem` interface, whose single method is `Eval`. The signature of `Eval` is a carefully chosen subset of the `Invoke` method of the `IDispatch` interface [13]. The implementation of `IScriptItem` makes sure that we can use normal function-call syntax for `Eval` in VB while the resulting outgoing call will still have exactly the three arguments as demanded by the `IScriptItem` interface. When using C or C++ we have to pass the three arguments explicitly.

On the Haskell side, dynamic values can be exported by the `addScriptItem` family of functions:

$$\begin{aligned} \text{addScriptItem}_{n,m} &:: \overbrace{(\dots, \text{Variant } a, \dots)}^{n>0}, \overbrace{(\dots, \text{Variant } b, \dots)}^m \\ &\Rightarrow \text{String} \\ &\rightarrow (\dots \rightarrow a \rightarrow \dots \rightarrow \text{IO } (\dots, b, \dots)) \\ &\rightarrow \text{IO } () \end{aligned}$$

Note the similarity between `addScriptItem` and `onEvent`, both of which make a Haskell value callable by the external world.

The following Visual Basic example loads the Haskell module `Fac.hs` and then computes a factorial in Haskell to leverage on Haskell's type `Integer` of infinite precision integers:

```
Dim Script As IScript
Set Script = CreateObject("HugsScript")
Script.Load("Fac.hs")
Set Fac = Script.GetScriptItem("Fac")
MsgBox (Fac.Eval(1000))
```

The Haskell module `Fac.hs` exports the standard tail-recursive factorial under the name `"Fac"`³:

```
main :: IO ()
main =
  do{ addScriptItem_1_1 "Fac" fac }
```

³ values of type `Integer` are unmarshalled as strings.

```

fac :: Integer -> Integer
fac =
  let{ fac m = \n ->
        case n of
          { 0 -> m
            ; n -> fac (m*n) (n-1)
          }
      } in (\n -> fac 1 n)

```

6 Conclusions

Most computer scientists dismiss declarative languages as naive, elegant curiosities that are largely irrelevant to the work-a-day world of programming. One of the primary reasons is that functional and logic languages have often lived in a closed world, or, at best, had awkward interfaces to the outside world that required significant investment to develop.

In our research program on the use of Haskell for scripting, we are addressing these challenges directly. First, we are exploiting the current popularity of COM and its Interface Description Language to get language independent descriptions of foreign functions with a minimal investment [5].

Second, we are exploiting the structure of the foreign functionality captured in the HTML object model. While there are some significant warts, for instance zero-argument callback functions and the fact that all interfaces are final, it is a useful structure and it supports a reasonable type-safe, natural coding style at the lowest level. In our prior experience it was always necessary to encapsulate foreign functionality in a carefully crafted layer of safe abstractions. In this context it is possible to use the raw, imported functionality directly. As we begin to explore other tools with automation interfaces (such as Visio and Microsoft Office applications) we are encouraged to find that “good object models” from an OO point of view are giving rise to usable automatically generated interfaces.

Third, we are reaping the benefits of explicitly monadic functional programming by passing “latently effectful computations” as values to event handlers and other functions. Ultimately scripting languages are IO behaviors. By having IO behaviors as first-class values, and not just the side effect of evaluation, we are able to support a compositional style of abstraction that is not possible in traditional scripting languages. This benefit of using Haskell as a scripting language is featured in earlier work, in which we develop parallel and sequential combinators for animation behaviors of Microsoft Agents [8].

All these factors converge to make Haskell an excellent language for scripting in general, and scripting Dynamic HTML in particular. However, many challenges remain. We are currently working to improve the robustness of the implementa-

tion and to develop a greater body of experience in scripting HTML and other tools with this technology.

HaskellScript is available at the Haskell homepage: <http://www.haskell.org>.

Acknowledgments

We would like to thank Conal Elliott and Jeff Lewis for their extensive comments, and the PADL referees for their apt remarks and constructive suggestions for improvement. Unfortunately, lack of space prevented us from fitting their bill completely. Erik Meijer much appreciated the hospitality of the PacSoft group at OGI and the Computer Sciences Research Center at Bell Labs in the initial stages of writing this paper.

This work was partially supported by the US Air Force under contract F19628-96-C-0161 and by the Department of Defense.

References

- [1] Richard Bird. *Introduction to Functional Programming using Haskell (2nd edition)*. Prentice Hall, 1998.
- [2] David Chappel. *Understanding ActiveX and OLE*. Microsoft Press, 1996.
- [3] J. Peterson (editor) et. al. Report on the programming language Haskell version 1.4. <http://www.haskell.org/>, April 6 1997.
- [4] Lauren Wood et. al. Document Object Model (DOM) Level 1 Specification. <http://www.w3.org/TR/REC-DOM-Level-1>, October 1998.
- [5] Sigbjørn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. H/Direct: A Binary Foreign Language Interface to Haskell. In *Proceedings of ICFP'98*, 1998.
- [6] Scott Isaacs. *Inside Dynamic HTML*. Microsoft Press, 1997.
- [7] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *POPL 20*, pages 71–84, 1993.
- [8] Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM components from Haskell. In *Proceedings of ICSR5*, 1998.
- [9] SL Peyton Jones and J Launchbury. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.
- [10] Håkon Wium Lie and Bert Bos, editors. *Cascading Style Sheets*. Addison-Wesley, 1997.
- [11] John Matthews, John Launchbury, and Byron Cook. Microprocessor Specification in Hawk. In *International Conference on Computer Languages*, 1998.
- [12] Erik Meijer. Server-side Scripting in Haskell. *Journal of Functional Programming*, Accepted for publication.
- [13] Microsoft Press. *Automation Programmers Reference*, 1997.
- [14] D. Ragget, Arnoud Le Hors, and Ian Jacobs. HTML 4.0 specification. <http://www.w3.org/TR/REC-html40>, December 1997.
- [15] Philip Wadler. Personal communication.
- [16] Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *LNCS*. Springer Verlag, 1995.

MCORBA: A CORBA Binding for Mercury

David Jeffery, Tyson Dowd, and Zoltan Somogyi

`{dgj,trd,zs}@cs.mu.oz.au`

Department of Computer Science and Software Engineering

University of Melbourne

Parkville, Victoria 3052, Australia

Abstract. MCORBA is a binding to the CORBA distributed object framework for the purely declarative logic/functional language Mercury. The binding preserves the referential transparency of the language, and has several advantages over similar bindings for other strongly typed declarative languages. As far as we know, it is the first such binding to be bidirectional; it allows a Mercury program both to operate upon CORBA components and to provide services to other CORBA components. Whereas the Haskell binding for COM maps COM interfaces onto Haskell types, MCORBA maps CORBA interfaces onto Mercury type classes. Our approach simplifies the mapping, makes the implementation of CORBA's interface inheritance straightforward, and makes it trivial for programmers to provide several different implementations of the same interface. It uses existential types to model the operation of asking CORBA for an object that satisfies a given interface but whose representation is unknown.

1 Introduction

Purely declarative programming languages have many benefits over imperative languages. Modern declarative programming languages are expressive, efficient, have clear semantics and are amenable to aggressive optimisation, which makes them very attractive tools for large classes of problems.

For declarative languages to be used extensively in real-world programs they cannot simply offer a solution (no matter how elegant) to a particular problem in isolation, nor can they demand exclusive use of one programming language. Declarative languages need to be able to be used as a part of a larger system, or their benefits to a particular subsystem will be ignored in face of the larger concerns of the system. Declarative languages must therefore include comprehensive interfaces with existing languages and systems if they are to see widespread adoption. Put simply, interoperability needs to have high priority in the language design.

We believe the design of the external interfaces of declarative languages should be guided by the following principles:

- The purity and expressiveness of the language should be maintained.
- The natural style of the language should not be unduly restricted (but can certainly be augmented) by an external interface.
- The interface should be as seamless as possible, unless visible seams have some practical benefit.
- The interface should be bi-directional, allowing code written in the declarative language to act both as provider and as consumer of services.

Many other languages and systems have had to address similar interoperability issues in recent years. This has led to the development and refinement of new software development architectures, including a strong movement towards the development of systems based around components. Each component has a well defined interface but the implementation is relatively unconstrained – the tools, techniques and languages used for each component can be chosen based upon their merits, rather than the demands of the rest of the system.

Component based systems are a scaled up version of programs based on abstract module interfaces, and like their smaller cousins, they come in procedural and object oriented flavours. Procedural systems are based on procedure calls from one component to another, whereas object oriented systems view each component as an object and pass messages between objects.

While at first these systems mainly put components together to form systems on a single machine, the growth of networking and particularly the Internet has led to the widespread use of distributed component based development, with systems being comprised of components connected via local or wide area networks. This involves either remote procedure calls, or distributed object systems invoking methods over a network.

Component based systems are an excellent opportunity to use declarative languages – their strengths can be put to work in isolation from the concerns of other components. This allows declarative languages to be adopted in a piecemeal fashion, which is an important consideration when programmers must be trained and familiarized with new techniques.

One of the most popular component based systems is CORBA, the Common Object Request Broker Architecture [2]. CORBA is a distributed object framework created and maintained by the Object Management Group (OMG) which represents the input of a large consortium of companies. CORBA is based around a standard Interface Definition Language (IDL). It uses Object Request Brokers (ORBs) to act as an object “bus”, which allows easy local or remote communication between objects.

This paper describes *MCORBA*, a CORBA binding for Mercury, a purely declarative logic/functional programming language that is designed for general purpose, large-scale, real-world use. This binding uses type classes and existential types, two language constructs recently added to the Mercury implementation, to provide a natural, expressive, and yet still purely declarative interface to CORBA. Although *MCORBA* is still only a prototype that does not yet address the full functionality of CORBA objects, we have already used it both to operate

upon existing CORBA components from Mercury and to implement CORBA components in Mercury.

H/Direct [1], an interface from Haskell to COM, Microsoft's Component Object Model [10] which is a CORBA competitor, uses some similar techniques to this paper. The lack of support for existential types in Haskell requires H/Direct to model component interfaces as types, The MCORBA approach, which we introduce in this paper, is to model component interfaces as type classes, which we believe to be significantly more natural.

Section 2 gives a brief overview of Mercury. Section 3 introduces CORBA and explains how we interface between CORBA and Mercury. The low-level details of the interface are explored in section 4.

2 Overview of Mercury

Mercury is a pure logic/functional programming language designed explicitly to support teams of programmers working on large application programs. In this section, we will briefly introduce the main features of Mercury, these being its module, mode, determinism and type systems, including two important recent additions to the type system: type classes and existential types. We will use the following simple program as the example for this introduction.

```
:- module example.

:- interface.
:- import_module io.
:- pred main(io:state, io:state).
:- mode main(di,uo) is det.

:- implementation.

main(S0, S) :-
  io:read_line(Result, S0, S1),
  (
    Result = ok(CharList),
    string:from_char_list(CharList, String),
    io:write_string("The input was: ", S1, S2),
    io:write_string(String, S2, S)
  ;
    Result = error(_),
    io:write_string("Some error occurred.\n", S1, S)
  ;
    Result = eof,
    S = S1
  ).
```

Syntax. As this example shows, the syntax of Mercury is very similar to the syntax of Prolog. Identifiers starting with upper case letters represent variables;

identifiers starting with lower case letters represent predicate or function symbols. Facts and rules comprise the code of the program, while terms starting with ‘:-’ are declarations. Mercury has many more kinds of these than Prolog.

Modules. Some of these declarations are part of the module system. In Mercury, every program consists of one or more modules. Each module defines one or more entities (e.g. types and predicates). Some of these entities are exported from the module; others are private to the module. The declarations of the former are in the interface section of the module while the declarations of the latter are in the implementation section. The definitions, where these are separate from the declarations (as they are for predicates and abstract types) are all in the implementation section. If a module wants to refer to any entity exported from another module, it must first import that module. The name of any entity may (and in case of ambiguity, must) be module qualified, i.e. prefixed by the name of its defining module and a colon.

In this example, the interface section contains the declaration of the predicate ‘**main**’, while the implementation section contains its code. The ‘:- **pred**’ declaration states that **main** has two arguments, which are both of the type **state** defined in module **io**. Since the interface section refers to this entity from module **io**, it must and does import module **io**.

Recent versions of Mercury also support nested modules.

Types. Mercury has a strong Hindley-Milner style type system with parametric polymorphism and higher-order types. ‘:- **pred**’ declarations specify the types of the arguments of predicates. Types themselves are defined like this:

```
:- type io:result(T) ---> ok(T) ; error(io:error) ; eof.
```

This defines the parametric type constructor **io:result** by stating that values of type **io:result(T)**, where **T** is a type variable standing in for an arbitrary concrete type, can be bound to one of the three function symbols (data constructors in the terminology of functional programming) **ok/1**, **error/1** and **eof/0** (numbers after the slash denote arities). If the value is **ok**, the type of its argument will be the concrete type bound to **T**, while if the value is **error**, its argument will be of type **io:error**.

Modes. Besides its type declaration, every predicate has a mode declaration, which says which arguments are input, and which are output. (A predicate may have more than one mode, but none of the predicates we discuss in this paper do.) Further, some mode declarations encode information about uniqueness. If the mode of a given argument of a given predicate is **out**, this represents an assertion by the programmer that the argument will be instantiated by the predicate. If the mode is **uo**, which stands for *unique output*, this represents a further assertion that the value returned to the caller will not have any other references pointing to it. Similarly, the mode **in**, represents an assertion that the argument will be instantiated by the caller. The mode **di**, which stands for *destructive input*, represents a further assertion that the value passed by the caller for this argument does not have any other references pointing to it. Therefore the predicate may destroy the value once it has finished using it.

The Mercury runtime system invokes the predicate `main/2`, which is analogous to the function `main()` in C, with a unique reference to an `io:state`, and requires `main` to return a unique reference to another `io:state`. `io:state` is a type representing the state of the world. It is an abstract type, whose name is exported from module `io` but whose definition is not, which means that only predicates in module `io` can manipulate values of type `io:state`. Predicates in other modules cannot do anything with such values except give them as arguments in calls to predicates in module `io`. By design, all these predicates have two arguments of type `io:state`, one whose mode is `di` and one whose mode is `uo`. Together, these constraints ensure that at any point in the forward execution there is exactly one live `io:state`. (The `io:state` arguments threaded through the program are usually hidden by definite clause grammar (DCG) notation.)

Determinism. To allow the predicates in module `io` to update the state of the world destructively without compromising referential transparency, we must make sure that versions of the `io:state` that have been destructively updated cannot become live after backward execution either. This requires us to reason about where backtracking may occur in the program. To this end, we associate with each mode of each predicate a determinism, which puts lower and upper bounds on the number of times the predicate may succeed when called in that mode. The determinism ‘`det`’ asserts that the predicate will succeed exactly once; ‘`semidet`’ asserts that it will succeed at most once; ‘`multi`’ asserts that it will succeed at least once; and ‘`nondet`’ means that it may succeed any number of times.

The Mercury compiler can prove that `main` above succeeds exactly once, and that backtracking can never cause obsolete `io:states` to become live, by noting that all the predicates called by `main` are declared `det`, that the unification `S = S1` acts as an assignment and is therefore `det`, and that since the type of `Result` is `io:result(list(char))`, exactly one of the three unifications involving `Result` will succeed regardless of what value `io:read_line` gives to `Result`. The Mercury compiler is required to prove *all* the assertions contained in `:- pred` and `:- mode` declarations.

Type classes. Mercury has always supported parametric polymorphism (unconstrained genericity). Recent versions of Mercury also support constrained genericity in the form of type classes [5], a language construct we borrowed from functional languages which is similar to Java’s interfaces. A ‘`:- typeclass`’ declaration such as

```
:- typeclass printable(T) where [
    pred print(T, io:state, io:state),
    mode print(in, di, uo) is det,
].
```

introduces a new type class, in this case the type class `printable/1`, and gives the signatures of the methods that must be implemented on a type if that type is to be considered a member of that type class. Specific types can then be declared to be members of this type class by giving implementations for the methods listed in the type class declaration. For example, given the predicate

```
:- pred io:write_int(int, io:state, io:state).
:- mode io:write_int(in, di, uo) is det.
```

the instance declaration

```
:- instance printable(int) where [
    pred(print/3) is io:write_int
].
```

makes the `int` type a member of the `printable/1` type class. This in turn makes it possible to pass `ints` where a printable type is needed. For example, the declaration of the predicate

```
:- pred print_list(list(T), io:state, io:state) <= printable(T).
:- mode print_list(in, di, uo) is det.
print_list([], S, S).
print_list([T | Ts], S0, S) :-
    print(T, S0, S1),
    print_list(Ts, S1, S).
```

has a *type class constraint* which requires the first argument to be a list of printable items because it invokes the `print` method on each element of this list.

Existential types. The language features we have discussed so far all require the caller to know the concrete types of all arguments. However, sometimes one wants a predicate to return a variable whose type is constrained to be in a particular type class but whose concrete type is unknown to the caller. Existential types provide a solution to this problem [7]; a type variable which is existentially quantified is bound by the *callee* of a predicate. For example, the predicate

```
:- some [T] pred read_printable(T, io:state, io:state)
    => printable(T).
:- mode read_printable(out, di, uo) is det.
```

reads and returns a data item whose type the caller does not know, but on which all the operations of the type class `printable` are defined.

For a detailed explanation of Mercury's type classes, see [4]; for more information on the rest of Mercury, e.g. functional syntax, please see the language reference manual [3].

3 Mercury Binding for CORBA

The Common Object Request Broker Architecture (CORBA) is a standard designed to tackle the problem of interoperability between software systems. It allows applications and components to communicate, whether or not they are executing in the same address space, whether or not they are executing on the same machine, and whether or not they are written in the same language.

Communication between software components in a system using CORBA is handled by an Object Request Broker (ORB). The ORB is responsible for establishing and maintaining communication between objects. The ORB is “middleware” through which a client can transparently invoke operations on an object, regardless of the location or implementation of the object. Different ORB implementations may communicate in different ways, so the CORBA 2.0 specification [2] includes a protocol for communication between ORBs, the Internet Inter-ORB Protocol (IIOP).

The interfaces of CORBA components are described in OMG’s Interface Definition Language (IDL). An IDL specification defines a set of *interfaces*, each of which is the abstract “signature” of an object. Each interface includes a set of operations and attributes. (Attributes are equivalent to a pair of operations, one that sets the attribute to a value and one that retrieves the value, so we will not mention them further.) For each operation, the interface defines the types of the parameters, and for each parameter, whether the parameter is an input (*‘in’*), an output (*‘out’*) or both (*‘inout’*). (This level of detail in the description is necessary for strongly typed languages such as Ada, Eiffel and Mercury, and even for weakly typed languages such as C when different components reside in different address spaces or on different machines.) For example, the following IDL specification:

```
interface CallBack {
    void NewMessage(in string msg);
};
```

defines an interface which has one operation, a one-argument procedure taking a string as input.

A CORBA binding for a particular language provides a way for code written in that language to operate on objects whose interface is described in IDL, and to implement objects whose interface is described in IDL. This binding takes the form of a tool that takes an IDL specification and automatically generates *stubs* and *skeletons* in the target language. When a client written in the target language wants to invoke one of the operations of a CORBA object, it invokes the corresponding stub routine. The stub routine acts as a proxy; it asks the ORB to invoke the same operation on the real object, regardless of its location and implementation language. The ORB communicates with the real object through the skeleton code, which invokes a method of the object when it receives a corresponding request from the ORB.

Standard mappings from IDL are defined for many programming languages including Java, C, C++, and Cobol. For C++, the stub will be a class that passes messages to its real implementation via the ORB. A user who requests an object from the ORB will be given an instance of the stub class, and will be able use it as if the class were implemented locally. The skeleton is an abstract class from which the user must derive a concrete class. Instances of the derived class can then be registered with the ORB so clients can use them, or be passed as parameters to other objects.

This paper may be seen as (the beginnings of) a Mercury binding for CORBA. This binding consists of a run-time library (called `corba`), and a tool (called `mcorba`) which generates Mercury stubs and skeletons from IDL specifications.

`mcorba` generates a Mercury type class for each IDL interface. For each operation of the interface, the type class has a method. The method will have two pairs of arguments with modes `di` and `uo`, to denote the state of the object and the state of the external world before and after the invocation of the method. The types and modes of the remaining arguments will correspond to the types and modes of the arguments of the IDL operation, with the exception that for `inout` parameters, we generate two arguments, with modes `in` and `out`.

From the IDL specification above, `mcorba` would generate

```
:- typeclass callback(T0) where [
    pred newmessage(T0, T0, corba:string, io:state, io:state),
    mode newmessage(di, uo, in, di, uo) is det
].
```

This type class serves two purposes:

- The ORB can give a client an object which belongs to the type class so that the client can invoke the appropriate operations on it (and no others).
- If a Mercury value is of a type which is an instance of the type class, it may be registered with the ORB as an object that satisfies the relevant interface.

We will now examine these two purposes in greater detail.

3.1 The Forwards Interface: Mercury Calls CORBA

The generation of type classes for IDL interfaces, outlined above, is sufficient for a Mercury program to invoke operations on a CORBA object. However, we also need a way of getting a handle on such an object from the ORB. CORBA provides several methods for doing this. Each method is implemented by a predicate in the `corba` module. The predicate

```
:- some [T] pred corba:resolve_object_name(orb, string, string, T,
    io:state, io:state) => corba:object(T).
:- mode corba:resolve_object_name(in, in, in, uo, di, uo) is det.
```

uses the COS (Common Object Services) naming service, through which a client may request an object by name. Note that the client does not request an object of a particular interface type, but merely a generic CORBA object. Since the argument is existentially typed, the implementation of this object is hidden from the client, but it is known to be connected to the ORB since the predicates of the `corba` module guarantee that only such objects satisfy the `corba:object` type class constraint. It is then the client's responsibility to ask the ORB to "narrow" this object into the required interface. It can do so by calling the `narrow` predicate that `mcorba` generates for the desired IDL interface along with

the type class. As this operation has the same name for each IDL interface, the code generated for each interface is placed inside a separate Mercury module.

We will show how this is done using as an example a “chat” server with three operations: sending a message to the server for distribution to all registered clients, registering a client, and removing a client. The IDL specification is:

```
interface Chat {
    void SendMessage(in string msg);
    void RegisterClient(in Callback obj, in string name);
    void RemoveClient(in Callback obj, in string name);
};
```

From this, mcorba generates the following type class:

```
:- typeclass chat(T0) where [
    pred sendmessage(T0, T0, corba:string, io:state, io:state),
    mode sendmessage(di, uo, in, di, uo) is det,
    pred registerclient(T0, T0, T1, corba:string, io:state,
        io:state) <= (callback(T1), corba:object(T1)),
    mode registerclient(di, uo, in, in, di, uo) is det,
    pred removeclient(T0, T0, T1, corba:string, io:state,
        io:state) <= (callback(T1), corba:object(T1)),
    mode removeclient(di, uo, in, in, di, uo) is det
].
```

Once again, there is a type class method corresponding to each IDL interface operation. This time, the `RegisterClient` and `RemoveClient` methods both have arguments which are objects satisfying IDL interfaces. The corresponding arguments of the type class methods are constrained to belong to both the `callback` and `corba:object` type classes, since the method may invoke specific `Callback` or generic CORBA operations on that argument.

The automatically generated `narrow` operation for the `Chat` interface has the signature:

```
:- type corba:narrow_result(U, N) ---> narrowed(N) ; unnarrowed(U).

:- some [C] pred chat:narrow(0, corba:narrow_result(0, C))
    => (chat(C), corba:object(C)) <= corba:object(0).
:- mode chat:narrow(di, uo) is det.
```

The `narrow` operation may or may not be successful; the CORBA object may or may not actually satisfy the required IDL interface. The ORB finds out whether it does or not. If it does, `narrow` will bind the second argument to `narrowed(Chat)`, where `Chat` is a value which belongs to the `chat` type class as well as `corba:object`. If not, it binds the second argument to `unnarrowed(Obj)`, where `Obj` is the original non-narrowed value.

The following code fragment shows how a Mercury program can use the library predicate `corba:resolve_object_name` and the automatically generated

predicate `chat:narrow` to connect to a CORBA chat server. The program can then send the lines of text that it reads from the user to the chat server. (Note that this code uses standard DCG notation to hide two parameters. The curly braces denote goals which do not use the hidden parameters; all other goals have two hidden `io:state` arguments threaded through them.)

```
:- pred sender(corba:orb, io:state, io:state).
:- mode sender(in, di, uo) is det.
sender(Orb) -->
    corba:resolve_object_name(Orb, "Chat", "Server", Obj),
    { chat:narrow(Obj, Narrowed) },
    (
        { Narrowed = narrowed(Chat) },
        sender_loop(Chat)
    ;
        { Narrowed = unnarrowed(_OriginalObj) },
        io:write_string("Couldn't narrow object\n")
    ).

:- pred sender_loop(T, io:state, io:state) <= chat(T).
:- mode sender_loop(di, di, uo) is det.
sender_loop(Chat0) -->
    io:read_line(Res),
    (
        { Res = ok(CharList) },
        { string:from_char_list(CharList, String) },
        sendmessage(Chat0, Chat, String),
        sender_loop(Chat)
    ;
        { Res = error(_Error) },
        io:write_string("Some kind of error occurred\n")
    ;
        { Res = eof }
    ).
```

3.2 The Backwards Interface: CORBA Calls Mercury

Mercury objects can be used to implement a CORBA object by providing an instance of the corresponding type class. For example, we can implement a `Chat` server as follows¹:

¹ Mercury does not yet support existentially quantified components of data structures, so we cannot store the callbacks in a list. We have instead represented each callback as a `callback_object`, and used Mercury's C interface to implement two predicates:

- `construct_callback_object`, which takes an arbitrary value in type classes `callback` and `corba:object` and produces a value of type `callback_object`.

```

:- type our_server ---> clients(list(callback_object))
:- instance chat(our_server) where [
    pred(sendmessage/5) is our_sendmessage,
    pred(registerclient/6) is our_registerclient,
    pred(removeclient/6) is our_removeclient
].

:- pred our_sendmessage(our_server, our_server, string,
    io:state, io:state).
:- mode our_sendmessage(di, uo, in, di, uo) is det.
our_sendmessage(clients([]), clients([]), _Msg --> []).
our_sendmessage(clients([CBObj0 | Rest0], clients([CBObj | Rest]),
    Msg) -->
    { deconstruct_callback_object(Obj0, CBObj0) },
    newmessage(Obj0, Obj, Msg),
    { construct_callback_object(Obj, CBObj) },
    our_sendmessage(clients(Rest0), clients(Rest), Msg).

```

If a type has been made an instance of a type class which corresponds to an IDL interface, we need to be able to tell the ORB that a particular object of that type is ready to service requests. We achieve this by generating another predicate for each generated type class. The `is_ready` predicate notifies the ORB that a Mercury value is ready to service requests. For the `Chat` example, we would get:

```

:- some [T2] pred chat:is_ready(corba:object_adaptor, T1, T2,
    io:state, io:state)
    => (chat(T2), corba:object(T2)) <= chat(T1).
:- mode chat:is_ready(in, di, uo, di, uo) is det.

```

A CORBA construct called an *object adaptor* keeps track of which objects are ready to service requests. A given ORB may have more than one object adaptor, so the caller of `chat:is_ready` must choose which object adaptor the object connects to.

With this machinery, our chat server above can make its services available as simply as this:

```

:- pred start_server(corba:orb, corba:object_adaptor,
    io:state, io:state).
:- mode start_server(in, in, di, uo) is det.
start_server(Orb, ObjectAdaptor) -->
    { Server = clients([]) },
    chat:is_ready(ObjectAdaptor, Server, ReadyServer),
    corba:bind_object_name(Orb, "Chat", "Server", ReadyServer),
    corba:implementation_is_ready(ObjectAdaptor).

```

-
- `deconstruct_callback_object`, which takes a `callback_object` and produces an existentially typed value in type classes `callback` and `corba:object`.

The call to `bind_object_name` registers the object with the name server. The call to `implementation_is_ready` hands control over to the event loop of the object adaptor, which listens for operations on any of the objects that have been registered with the adaptor, and invokes the corresponding methods. Each method invocation will generate a new version of the state of the object, which becomes the state of the object for the next method invocation. These states are not visible from outside of the event loop; the only *visible* effect of the methods is on the I/O state. The I/O state arguments of `implementation_is_ready` (here hidden using DCG notation) capture these effects, just as the I/O state arguments of `io:write_string` do. Therefore method invocations are purely declarative for the same reasons that I/O operations are.

3.3 Other Features of CORBA

The CORBA binding we have described above has touched upon only a few of CORBA's features. We will now briefly describe how we handle some other CORBA features.

- One IDL interface may inherit from another. For example, with:

```
interface sub : super {
    ...
};
```

all of the operations available on `super` are also available on `sub`. We map IDL interface inheritance directly onto type class inheritance. In Mercury, type class inheritance is denoted by a type class declaration itself having type class constraints. In this example:

```
:- typeclass sub(T) <= super(T) where [
    ...
].
```

any member of type class `sub` must also be a member of `super`. (Neither IDL nor Mercury allows any kind of implementation inheritance.)

- The CORBA IDL includes several basic types. Many of these (e.g. `double`, `int`, `char`, `string` and `boolean`) can have their values mapped one-to-one to the values of a Mercury type, which allows us to map the IDL type directly to the Mercury type. Some other IDL types (e.g. `float` and `short`), whose values are a subset of the values of a Mercury type, can be handled by mapping them to the Mercury type and having code automatically generated by `mcorba` check their values before giving them to CORBA. The remaining types are harder to handle. We can map them to new, abstract and non-native Mercury types that can then have the appropriate operations defined on them, but whether users will find such an arrangement convenient enough remains a matter for further experimentation.

- The CORBA IDL also includes several ways to build more complex types. Some of these (e.g. enumerations, structures, unbounded length sequences and discriminated unions in which the discriminant is an enumeration) can be mapped to Mercury in a natural manner, though some of these mappings may imply significant conversion costs. The rest (e.g. bounded length sequences and discriminated unions with integer discriminants) do not have obvious direct mappings to Mercury (or to Haskell either [1]), and finding a convenient way to represent them in Mercury is also future work.
- An IDL specification may declare sub-modules which introduce a new (nested) namespace. For each such module we create a Mercury sub-module.
- Constants can also be defined in a specification. For each such constant, we define a function with no input arguments but with a result corresponding to the constant.
- IDL also includes exceptions. We could handle these by mapping CORBA methods onto Mercury predicates that return either an exception indication or their actual result, (the way we handled the `narrow` operation in section 3.1), but we expect that most users would find these too cumbersome. We cannot use the obvious alternative solution, mapping CORBA exceptions to Mercury exceptions, until exceptions formally become part of the Mercury language.
- CORBA allows dynamic lookup of available operations through its Dynamic Invocation Interface (DII), and dynamic creation of operations through the Dynamic Skeleton Interface (DSI). Incorporating DII and DSI into MCORBA is a topic for future work.

4 Implementation

The primary goals of the initial implementation were simplicity and expediency. We wanted to allow easy experimentation with the mapping from IDL to Mercury, and permit handwritten code to be substituted for machine generated code when necessary. We therefore decided that instead of writing a binding which talks to an ORB directly (using IIOP for example), we would simply build on top of an existing binding. The binding we chose to build upon is omniORB 2.0 [8], whose `omniidl2` tool generates stubs and skeletons in C++. Besides generating Mercury stubs and skeletons, `mcorba` also generates the glue that joins the Mercury stub to the C++ stub, and the Mercury skeleton to the C++ skeleton. Since omniORB 2.0 conforms to the CORBA 2.0 standard [2], which specifies a standard C++ binding, our implementation can easily be ported to other ORBs with C++ bindings.

Building a Mercury layer on top of a C++ binding is somewhat complicated by the fact that the current Mercury implementation has an interface only to C, not to C++. Therefore `mcorba` generates C functions that provide a simple interface to C++ code. Since these C functions complicate the explanation of the implementation, in the following discussion we will assume that Mercury can directly call C++. (Building a Mercury layer on top of a C binding would avoid

these complications, so when a satisfactory C binding becomes available we will examine it.) We will also simplify the example code by skipping the required error checking and type casts.

4.1 The Forwards Interface: Mercury Uses CORBA Components

The stub predicate generated by `mcorba` for a given IDL operation will invoke the corresponding method on the C++ stub object. This requires Mercury code to remember pointers to these objects. When the program calls a predicate such as `is_ready`, which returns a handle on the underlying C++ object, the actual return value will be a pointer to the C++ object. This pointer is a value in the type `corba:cppobject`, whose implementation is known only to the MCORBA system. Since values of type `corba:cppobject` are only ever returned as existentially quantified type arguments belonging to the `corba:object` type class, user code can never break the abstraction.

For each operation in an IDL specification, `mcorba` generates a predicate implemented in C++. This predicate casts the `cppobject` to a pointer to the right C++ class, invokes the requested method, and then creates the new versions of the object and the I/O state.

```
:- pred sendmessage_impl(corba:cppobject, corba:cppobject,
    corba:string, io:state, io: state).
:- mode sendmessage_impl(di, uo, in, di, uo) is det.
:- pragma c_code(sendmessage_impl(Obj0::di, Obj::uo,
    Msg::in, IO0::di, IO::uo), may_call_mercury, "
    Chat_ptr chat = cast_to_chat(Obj0);
    chat->SendMessage(Msg);
    *Obj = cast_from_chat(chat);
    update_io(IO0, IO);
").
```

Given one of these predicates for each method, `mcorba` creates an instance of the chat type class:

```
:- instance chat(corba:cppobject) where [
    pred(sendmessage/5) is sendmessage_impl,
    pred(registerclient/6) is registerclient_impl,
    pred(removeclient/6) is removeclient_impl
].
```

4.2 The Backwards Interface: Mercury as a CORBA Component

Any Mercury type that is an instance of the generated type class for an interface can be used as an implementation of that interface. For the ORB to use the Mercury implementation there needs to be a C++ implementation, and a layer that allows the C++ implementation to call Mercury. To do this, the C++

implementation needs to know how to invoke a given method on a given Mercury object. In C++ this would be done using a virtual function table. Mercury has a similar construct called a *type class dictionary*. A Mercury predicate that manipulates an object of a given type class will have access to a type class dictionary that specifies which predicates implement the methods of the type class on that object. (A type class declaration defines the layout of dictionaries for that type class; an instance declaration specifies its contents.)

`omniidl2` generates an abstract base class for every IDL interface. `mcorba` derives a subclass from each such base class. The derived class simply stores the Mercury object and its type class dictionary as attributes. For our chat example, `omniidl2` generates the base class `_sk_Chat`, while `mcorba` generates the derived class `Chat_i`:

```
class Chat_i : public virtual _sk_Chat {
public:
    Chat_i(void *mcorba_tcdict, void *mcorba_mobj) {
        mcorba_typeclass_dict = mcorba_tcdict;
        mcorba_mercury_object = mcorba_mobj;
    }
    virtual ~Chat_i() {}
    void SendMessage(const char *);
    void RegisterClient(class Callback *, const char *);
    void RemoveClient(class Callback *, const char *);
private:
    void *mcorba_typeclass_dict;
    void *mcorba_mercury_object;
};
```

The methods of this C++ class call the methods of the corresponding Mercury type class. This involves looking up the relevant method in the type class dictionary, and calling it with the appropriate arguments. The Mercury type class implementation already creates code to do the lookup and parameter passing, so we simply export this code using the `pragma export` directive to allow it to be called from C (or, in our case, C++):

```
:- pragma export(sendmessage(di, uo, in, di, uo),
    "chat__sendmessage_method").
```

The implementation of the C++ method can simply call the exported C function with the stored type class dictionary, the original value of the object, and the input arguments of the method. The function will return the updated value of the object and the output arguments of the method through reference parameters.

```
void Chat_i::SendMessage(const char *msg)
{
    chat__sendmessage_method(mcorba_typeclass_dict,
        mcorba_mercury_object, &mcorba_mercury_object, msg);
}
```

Note that the C++ code does not know anything about I/O states, and that the two arguments of type `io:state` are missing. This is possible because Mercury uses values of this type only to enforce referential transparency; they do not actually have any representation.

Each invocation of the `mcorba`-generated `chat:is_ready` predicate creates an instance of the `Chat_i` class and calls the C++ `_sk_Chat::_obj_is_ready` method to register it with the ORB.

5 Conclusions and Further Work

The main contribution of this paper is showing how IDL interfaces can be mapped onto type classes. Since existential types are an integral part of this approach but are not (yet) part of the standard language specification for Haskell, the Haskell binding for COM [6] could not take this approach, and mapped IDL interfaces onto types instead. The advantages of our approach are that:

- Type classes provide a significantly more natural means of modeling CORBA interfaces. In particular, it is very easy for a program to provide more than one implementation of an interface by providing more than one `instance` declaration for it.
- Inheritance of one interface by another comes “for free”. There is no need to explicitly convert from an inherited-by to an inherited-to interface; the type class constraint mechanism handles this automatically.

Our approach should be easily adaptable to other logic and/or functional languages that have both type classes and existential types, as we expect Haskell 2 will when it arrives.

Our approach also has an advantage over C++ bindings. When an argument of a CORBA operation is another CORBA object, C++ bindings check that the actual parameter satisfies the right interface, but do not check that it has been registered with an ORB. The type class method generated by `mcorba` for the operation requires the corresponding argument to be a member of the `corba:object` type class, which guarantees that any attempt to pass an unregistered object as the actual parameter will be caught at compile time.

We have found our approach to the implementation, which is the use of a thin layer on top of an existing C++ binding, to be very effective. It can be implemented relatively quickly using a simple foreign language interface, and the resulting binding should be easily portable among different ORB implementations for C++.

MCORBA can already generate all the stubs and skeletons required by the chat server. Nevertheless, at the moment it is only a prototype. It doesn’t yet handle CORBA exceptions or several CORBA IDL types, and we have not yet started to work on fitting CORBA’s Dynamic Skeleton/Invocation Interfaces into our framework. We are working on lifting these limitations.

MCORBA is available from <http://www.cs.mu.oz.au/mercury>. We would like to thank the Australian Research Council for their support.

References

- [1] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. H/Direct: a binary foreign language interface for Haskell. In *Proceedings of the 1998 International Conference on Functional Programming*, September 1998.
- [2] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.0*. Object Management Group, 492 Old Connecticut Path, Framingham, MA 01701, U.S.A., July 1996.
- [3] Fergus Henderson, Thomas Conway, Zoltan Somogyi, and David Jeffery. The Mercury language reference manual. Technical Report 96/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1996.
- [4] David Jeffery, Fergus Henderson, and Zoltan Somogyi. Type classes in Mercury. Technical Report 98/13, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1998.
- [5] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: an exploration of the design space. In *Proceedings of the Haskell Workshop*, volume 788 of *Lecture Notes in Computer Science*. Springer Verlag, June 1997.
- [6] Simon Peyton Jones, Erik Meijer, and Daan Leijen. Scripting COM components from Haskell. In *Proceedings of the Fifth International Conference on Software Reuse*, June 1998.
- [7] Konstantin Läufer. Type classes with existential types. *Journal of Functional Programming*, 6(3):485–517, May 1996.
- [8] Sai-Lai Lo. *The omniORB2 User's Guide*. Olivetti and Oracle Research Laboratory, March 1997.
- [9] E. G. J. M. H. Nöcker, J. E. W. Smetsers, M. C. J. D. Eekelen, and M. J. Plasmeijer. Concurrent Clean. In *Proceedings of the Conference on Parallel Architectures and Languages Europe*, pages 202–219, Eindhoven, The Netherlands, June 1991.
- [10] Dale Rogerson. *Inside COM*. Microsoft Press, 1998.

Dead Code Elimination through Dependent Types^{*}

Hongwei Xi

Department of Computer Science and Engineering
Oregon Graduate Institute
P.O. Box 91000
Portland, OR 97291, USA
hongwei@cse.ogi.edu

Abstract. Pattern matching is an important feature in various functional programming languages such as SML, Caml, Haskell, etc. In these languages, unreachable or redundant matching clauses, which can be regarded as a special form of dead code, are a rich source for program errors. Therefore, eliminating unreachable matching clauses at compile-time can significantly enhance program error detection. Furthermore, this can also lead to significantly more efficient code at run-time.

We present a novel approach to eliminating unreachable matching clauses through the use of the dependent type system of DML, a functional programming language that enriches ML with a restricted form of dependent types. We then prove the correctness of the approach, which consists of the major technical contribution of the paper. In addition, we demonstrate the applicability of our approach to dead code elimination through some realistic examples. This constitutes a practical application of dependent types to functional programming, and in return it provides us with further support for the methodology adopted in our research on *dependent types in practical programming*.

1 Introduction

There is no precise definition of dead code in the literature. In this paper, we refer dead code as the code which can never be executed at run-time. Notice that this is essentially different from dead computation [1], that is, the computation producing values which are never used. For instance, in the following C code,

```
x = 1; x = 2;
```

the part `x = 1` is dead computation but not dead code since it is executed but its execution does not affect the entire computation. Also dead code is different from partially dead code, which is not executed only on *some* computation paths [10]. For instance, the part `y = 1` in the following C code is partially dead since it is not executed when `x` is not zero.

^{*} The research reported in this paper was supported in part by the United States Air Force Materiel Command (F19628-96-C-0161) and the Department of Defense.

```
if (x == 0) { y = 1; }
```

Pattern matching is an important feature in many functional programming languages such as Standard ML [12], Caml [16], Haskell [6], etc. A particular form of dead code in these languages is unreachable or redundant matching clauses, that is, matching clauses which can never be chosen at run-time. The following is a straightforward but unrealistic example of dead code in Standard ML.

```
exception Unreachable
fun foo l =
  case l of nil => 1 | _::_ => 1 | _ => raise Unreachable
```

The function `foo` is of type `'a list -> 'a list`. Clearly, the third matching clause in the definition of `foo` can never be chosen since every value of type `'a list` matches either the first or the second clause. This form of unreachable matching clauses can be readily detected in practice. For instance, if the above code is passed to the (current version of) SML/NJ compiler, an *error* message is reported describing the redundancy of the third matching clause. However, there are realistic cases which are much more subtle. Let us consider the following example.

```
exception ZipException
fun zip(nil, nil) = nil
  | zip(x::xs, y::ys) = (x, y)::zip(xs, ys)
  | zip _ = raise ZipException
```

The function `zip` is meant to zip two lists of *equal* length into one. This is somewhat hinted in the program since an exception is raised if two given lists are of different lengths. If `zip` is applied to a pair of lists of equal length, then the third matching clause in its definition can never be chosen for the obvious reason. Therefore, we can eliminate the third matching clause as long as we can guarantee that `zip` is *always* applied to a pair of lists of equal length. Unfortunately, it is *impossible* in Standard ML (or other languages with similar or weaker typing systems) to restrict the application of `zip` only to pairs of lists of equal length since its type system cannot distinguish lists of different lengths. This partially motivated our research on strengthening the type system of ML with dependent types so that we can formulated more accurate types such as the type of all pairs of lists of equal length.

The type system of DML (Dependent ML) in [20] enriches that of ML with a *restricted form* of dependent types. The primary goal of this enrichment is to provide for specification and inference of significantly more accurate information, facilitating both program error detection and compiler optimization. For instance, the following declaration in DML enables the type system to distinguish lists of different lengths.

```
datatype 'a list = nil | :: of 'a * 'a list
typeref 'a list of nat with
  nil <| 'a list(0)
  | :: <| {n:nat} 'a * 'a list(n) -> 'a list(n+1)
```

The declaration defines a (polymorphic) datatype `'a list` to represent the type of lists. This datatype is then indexed by a natural number, which stands for the length of a list in this case. The constructors associated with the datatype `'a list` are then assigned dependent types:

- `nil <| 'a list(0)` states that `nil` is a list of length 0, and
- `:: <| {n:nat} 'a * 'a list(n) -> 'a list(n+1)` states that `::` yields a list of length $n + 1$ when given a pair consisting of an element and a list of length n . Note that `{n:nat}` means that n is universally quantified over natural numbers, usually written as $\prod n : \text{nat}$.

Now the following definition of the `zip` function in DML guarantees that this function can only be applied to a pair of lists of equal length.

```
fun('a, 'b)
  zip_safe(nil, nil) = nil
| zip_safe(x::xs, y::ys) = (x, y)::zip_safe(xs, ys)
where zip_safe <| {n:nat} 'a list(n)*'b list(n)->('a * 'b) list(n)
```

The use of `fun('a, 'b)` is a recent feature of Standard ML [12], which allows the programmer to explicitly control the scope of the type variables `'a` and `'b`. The type of `zip_safe` is `{n:nat} 'a list(n) * 'b list(n) -> ('a * 'b) list(n)` which states that `zip_safe` can only accept a pair of lists of equal length and always returns a list of the same length. Notice that the `where` clause is a type annotation which must be provided by the programmer. If this function is applied to a pair of lists of unequal lengths, the code cannot pass the type-checking in DML and therefore is rejected. In this case, the programmer can certainly choose to use the previously defined `zip` if necessary in order to pass type-checking.

The programmer often knows that certain matching clauses can never be executed if a program is implemented correctly. Therefore, it can lead to program error detection at compile-time if we can verify whether these clauses can indeed be eliminated in an implementation. For example, when implementing an evaluator for the pure call-by-value λ -calculus, we know that we can never encounter a variable which is not bound to a closure during the evaluation of a *closed* λ -expression. As shown in Section 4.2, this can be readily verified by our approach. Eliminating dead code can also lead to more efficient execution at run-time. For instance, `zip_safe` need not check the tag of the second list in its argument since it is always the same as that of the first one, and this can contribute to 20% speedup on a Sun Sparc 20 running SML/NJ version 110. In general, our approach can be regarded as an example which supports that safety and efficiency can be *complementary*. These advantages yield some strong justification for our approach to eliminating unreachable matching clauses at compile-time.

We organize the paper as follows. In Section 2, we give some preliminaries on the core of DML, which provides all the machinery needed to establish the correctness of our approach to dead code elimination. We then present in Section 3 the derivation rules which formalize the approach, and prove the correctness of

these rules. This constitutes the main technical contribution of this paper. In Section 4, we use some realistic examples to demonstrate the applicability of our approach. The rest of the paper discusses some related work and then concludes.

2 Preliminaries

It is both infeasible and unnecessary to present in this paper the entire DML, which can be found in [20]. We refer the reader to [18] for an overview of DML. The essence of our approach will be fully captured in an core language $\text{ML}_0^H(C)$, which is a monomorphic extension of mini-ML with general pattern matching and universal dependent types.

Note that the omission of ML let-polymorphism in $\text{ML}_0^H(C)$, which is fully supported in DML, is simply for the sake of brevity of the presentation. Since $\text{ML}_0^H(C)$ parameterizes over a constraint domain C from which type index objects are drawn, we start with a brief introduction of the formation of a constraint domain.

2.1 Constraint Domains

We emphasize that the general constraint language itself is typed. In order to avoid potential confusion we call the types in the constraint language *index sorts*. We use b for base index sorts such as o for propositions and int for integers. A signature Σ declares a set of function symbols and associates with every function symbol an *index sort* defined below. A Σ -structure \mathcal{D} consists of a set $\mathbf{dom}(\mathcal{D})$ and an assignment of functions to the function symbols in Σ . A constraint domain $C = \langle \Sigma, \mathcal{D} \rangle$ consists of a signature Σ and a Σ -structure \mathcal{D} .

We use f for interpreted function symbols (constants are 0-ary functions), p for atomic predicates (that is, functions of sort $\gamma \rightarrow o$) and we assume that we have constants such as equality, truth values \top and \perp , conjunction \wedge , and disjunction \vee , all of which are interpreted as usual.

$$\begin{aligned} \text{index sorts} \quad \gamma &::= b \mid \mathbf{1} \mid \gamma_1 * \gamma_2 \mid \{a : \gamma \mid P\} \\ \text{index propositions } P &::= \top \mid \perp \mid p(i) \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \end{aligned}$$

Here $\{a : \gamma \mid P\}$ is the subset index sort for those elements of index sort γ satisfying proposition P , where P is an index proposition. For instance, nat is an abbreviation for $\{a : int \mid a \geq 0\}$, that is, nat is a subset index sort of int .

We use a for index variables in the following formation. We assume that there exists a predicate \doteq of sort $\gamma * \gamma \rightarrow o$ for every index sort γ , which is interpreted as equality. Also we emphasize that all function symbols declared in Σ must be associated with index sorts of form $\gamma \rightarrow b$ or b . In other words, the constraint language is first-order.

$$\begin{aligned} \text{index objects} \quad i, j &::= a \mid f(i) \mid \langle \rangle \mid \langle i, j \rangle \mid \mathbf{fst}(i) \mid \mathbf{snd}(i) \\ \text{index contexts} \quad \phi &::= \cdot_{\text{ind}} \mid \phi, a : \gamma \mid \phi, P \\ \text{index constraints} \quad \Phi &::= i \doteq j \mid \top \mid \Phi_1 \wedge \Phi_2 \mid P \supset \Phi \mid \forall a : \gamma. \Phi \mid \exists a : \gamma. \Phi \\ \text{index substitutions} \quad \theta &::= [] \mid \theta[a \mapsto i] \\ \text{satisfiability relation} \quad \phi &\models \Phi \end{aligned}$$

Note \cdot_{ind} is for the empty index context. We omit the standard sorting rules for this language for the sake of brevity. The satisfiability relation $\phi \models \Phi$ means that $(\phi)\Phi$ is satisfiable in the domain $\mathbf{dom}(\mathcal{D})$ of the Σ -structure \mathcal{D} , where $(\phi)\Phi$ is defined as follows.

$$\begin{aligned}
 (\cdot_{\text{ind}})\Phi &= \Phi \\
 (a : b)\Phi &= \forall a : b. \Phi \\
 (a : \gamma_1 * \gamma_2)\Phi &= (a_1 : \gamma_1)(a_2 : \gamma_2)\Phi[a \mapsto \langle a_1, a_2 \rangle] \\
 (\phi, \{a : \gamma \mid P\})\Phi &= (\phi)(a : \gamma)(P \supset \Phi) \\
 (\phi, P)\Phi &= (\phi)(P \supset \Phi)
 \end{aligned}$$

In this paper, we are primarily interested in the integer constraint domain, where the signature Σ_{int} is given in Figure 1 and the domain of Σ_{int} -structure is simply the set of integers. For instance, let $\phi = n : \text{nat}, a : \text{nat}, a + 1 \doteq n, 0 \doteq n$, then $\phi \models \perp$ holds in this domain since $(\phi)\perp$, defined as follows, is true.

$$\forall n : \text{int}. n \geq 0 \supset \forall a : \text{int}. a \geq 0 \supset (a + 1 \doteq n \supset (0 \doteq n \supset \perp))$$

Basically, $a + 1 \doteq n$ implies $n > 0$ since a is a natural number, and this contradicts $0 \doteq n$. In other words, the index context ϕ is inconsistent. This example will be used later.

$\Sigma_{\text{int}} =$	$\text{abs} : \text{int} \rightarrow \text{int}$	$\text{sgn} : \text{int} \rightarrow \text{int}$	$+$: $\text{int} * \text{int} \rightarrow \text{int}$
	$-$: $\text{int} * \text{int} \rightarrow \text{int}$	$*$: $\text{int} * \text{int} \rightarrow \text{int}$	$\text{div} : \text{int} * \text{int} \rightarrow \text{int}$
	$\text{min} : \text{int} * \text{int} \rightarrow \text{int}$	$\text{max} : \text{int} * \text{int} \rightarrow \text{int}$	$\text{mod} : \text{int} * \text{int} \rightarrow \text{int}$
	$<$: $\text{int} * \text{int} \rightarrow o$	\leq : $\text{int} * \text{int} \rightarrow o$	$=$: $\text{int} * \text{int} \rightarrow o$
	\geq : $\text{int} * \text{int} \rightarrow o$	$>$: $\text{int} * \text{int} \rightarrow o$	\neq : $\text{int} * \text{int} \rightarrow o$

Fig. 1. The signature of the integer domain

2.2 The Language $\text{ML}_0^H(C)$

Given a constraint domain C , the syntax of $\text{ML}_0^H(C)$ is given in Figure 2. The syntax is relatively involved because we must include general pattern matching in order to present our approach. This is an explicitly typed language since the dead code elimination we propose is performed when type-checking a program is done and an explicitly typed version of the program has been constructed.

We use δ for base type families, where we use $\delta(\langle \rangle)$ for an unindexed type. Also we use \cdot_{ms} , \cdot_{ind} and \cdot for empty matches, empty index context and empty context, respectively. The difference between values and value forms is that the former is only closed under value substitutions while the latter is closed under all substitutions. We write $e[i]$ for the application of an expression to a type index and $\lambda a : \gamma. e$ for index variable abstraction.

families	$\delta ::= (\text{family of refined datatypes})$
signature	$\mathcal{S} ::= \cdot_{\text{sig}} \mid S, \delta : \gamma \rightarrow *$ $\mid \mathcal{S}, c : \Pi a_1 : \gamma_1 \dots \Pi a_n : \gamma_n. \delta(i)$ $\mid \mathcal{S}, c : \Pi a_1 : \gamma_1 \dots \Pi a_n : \gamma_n. \tau \rightarrow \delta(i)$
types	$\tau ::= \delta(i) \mid \mathbf{1} \mid (\tau_1 * \tau_2) \mid (\tau_1 \rightarrow \tau_2) \mid (\Pi a : \gamma. \tau)$
patterns	$p ::= x \mid c[a_1] \dots [a_n] \mid c[a_1] \dots [a_n](p) \mid \langle \rangle \mid \langle p_1, p_2 \rangle$
matches	$ms ::= \cdot_{\text{ms}} \mid (p \Rightarrow e \mid ms)$
expressions	$e ::= x \mid \langle \rangle \mid \langle e_1, e_2 \rangle \mid c[i_1] \dots [i_n] \mid c[i_1] \dots [i_n](e)$ $\mid (\text{case } e \text{ of } ms) \mid (\text{lam } x : \tau. e) \mid e_1(e_2)$ $\mid \text{let } x = e_1 \text{ in } e_2 \text{ end} \mid (\text{fix } f : \tau. u)$ $\mid (\lambda a : \gamma. e) \mid e[i]$
value forms	$u ::= c[i_1] \dots [i_n] \mid c[i_1] \dots [i_n](u) \mid \langle \rangle \mid \langle u_1, u_2 \rangle$ $\mid (\text{lam } x : \tau. e) \mid (\lambda a : \gamma. u)$
values	$v ::= x \mid c[i_1] \dots [i_n] \mid c[i_1] \dots [i_n](v) \mid \langle \rangle \mid \langle v_1, v_2 \rangle$ $\mid (\text{lam } x : \tau. e) \mid (\lambda a : \gamma. v)$
contexts	$\Gamma ::= \cdot \mid \Gamma, x : \tau$
index contexts	$\phi ::= \cdot_{\text{ind}} \mid \phi, a : \gamma \mid \phi, P$
substitutions	$\theta ::= [] \mid \theta[x \mapsto e] \mid \theta[a \mapsto i]$

Fig. 2. The syntax for $\text{ML}_0^{\Pi}(C)$

We leave out polymorphism in $\text{ML}_0^{\Pi}(C)$ because polymorphism is largely orthogonal to the development of dependent types and has little effect on the dead code elimination presented in this paper. This tremendously simplifies the presentation.

In the rest of the paper, we assume that datatypes *intList* (for integer lists) and *intPairList* (for integer pair lists) have been declared with associated constructors *intNil*, *intCons* and *intPairNil*, *intPairCons*, respectively, and refined as follows.

```

intNil      : intList(0)
intCons     :  $\Pi n : \text{nat}. \text{int} * \text{intList}(n) \rightarrow \text{intList}(n + 1)$ 
intPairNil  : intPairList(0)
intPairCons :  $\Pi n : \text{nat}. (\text{int} * \text{int}) * \text{intPairList}(n) \rightarrow \text{intPairList}(n + 1)$ 

```

However, we will write *nil* for either *intNil* or *intPairNil* and *cons* for either *intCons* or *intPairCons* if there is no danger of confusion. The following expression *zipdef* in $\text{ML}_0^{\Pi}(C)$ corresponds a monomorphic version of the previously defined function *zip_safe* (we use *b* for an index variable here).

```

fix zip :  $\Pi n : \text{nat}. \text{intList}(n) * \text{intList}(n) \rightarrow \text{intPairList}(n).$ 
 $\lambda n : \text{nat}. \text{lam } l : \text{intList}(n) * \text{intList}(n).$ 
  case l of (nil, nil)  $\Rightarrow$  nil
          | (cons[a](x, xs), cons[b](y, ys))  $\Rightarrow$  cons[a]((x, y), zip[a](xs, ys))

```

Static Semantics We use $\phi \models \tau \equiv \tau'$ for the congruent extension of $\phi \models i \doteq j$ from index objects to types, which is determined by the following rules.

$$\begin{array}{c}
\frac{\phi \models i \doteq j}{\phi \models \delta(i) \equiv \delta(j)} \\
\frac{\phi \models \tau'_1 \equiv \tau_1 \quad \phi \models \tau_2 \equiv \tau'_2}{\phi \models \tau_1 \rightarrow \tau_2 \equiv \tau'_1 \rightarrow \tau'_2}
\end{array}
\qquad
\begin{array}{c}
\frac{\phi \models \tau_1 \equiv \tau'_1 \quad \phi \models \tau_2 \equiv \tau'_2}{\phi \models \tau_1 * \tau_2 \equiv \tau'_1 * \tau'_2} \\
\frac{\phi, a : \gamma \models \tau \equiv \tau'}{\phi \models \Pi a : \gamma. \tau \equiv \Pi a : \gamma. \tau'}
\end{array}$$

We start with the typing rules for patterns, which are listed in Figure 3. These rules are essential to the formulation of our approach to eliminating unreachable matching clauses. The judgment $p \downarrow \tau \triangleright (\phi; \Gamma)$ expresses that the index and ordinary variables in pattern p have the sorts and types in ϕ and Γ , respectively, if we know that p must have type τ .

$$\begin{array}{c}
\frac{}{x \downarrow \tau \triangleright (\cdot_{\text{ind}}; x : \tau)} \text{ (pat-var)} \qquad \frac{}{\langle \rangle \downarrow \mathbf{1} \triangleright (\cdot_{\text{ind}}; \cdot)} \text{ (pat-unit)} \\
\frac{p_1 \downarrow \tau_1 \triangleright (\phi_1; \Gamma_1) \quad p_2 \downarrow \tau_2 \triangleright (\phi_2; \Gamma_2)}{\langle p_1, p_2 \rangle \downarrow \tau_1 * \tau_2 \triangleright (\phi_1, \phi_2; \Gamma_1, \Gamma_2)} \text{ (pat-prod)} \\
\frac{\mathcal{S}(c) = \Pi a_1 : \gamma_1 \dots \Pi a_n : \gamma_n. \delta(i)}{c[a_1] \dots [a_n] \downarrow \delta(j) \triangleright (a_1 : \gamma_1, \dots, a_n : \gamma_n, i \doteq j; \cdot)} \text{ (pat-cons-wo)} \\
\frac{\mathcal{S}(c) = \Pi a_1 : \gamma_1 \dots \Pi a_n : \gamma_n. (\tau \rightarrow \delta(i)) \quad p \downarrow \tau \triangleright (\phi; \Gamma)}{c[a_1] \dots [a_n](p) \downarrow \delta(j) \triangleright (a_1 : \gamma_1, \dots, a_n : \gamma_n, i \doteq j, \phi; \Gamma)} \text{ (pat-cons-w)}
\end{array}$$

Fig. 3. Typing rules for patterns

We omit the rest of typing rules for $\text{ML}_0^\Pi(C)$, which can be found in [20]. The following lemma is at the heart of our approach to dead code elimination.

Lemma 1. (*Main Lemma*) Suppose that $p \downarrow \tau \triangleright \phi; \Gamma$ is derivable and $\phi \models \perp$ satisfiable. If $\cdot_{\text{ind}}; \cdot \vdash v : \tau$ is derivable, then v does not match the pattern p .

Proof. The proof proceeds by structural induction on the derivation of $p \downarrow \tau \triangleright \phi; \Gamma$. We present one interesting case where p is of form $c[a_1] \dots [a_n](p')$ for some constructor c . Then the derivation of $p \downarrow \tau \triangleright \phi; \Gamma$ must be of the following form.

$$\frac{\mathcal{S}(c) = \Pi a_1 : \gamma_1 \dots \Pi a_n : \gamma_n. (\tau' \rightarrow \delta(i)) \quad p' \downarrow \tau' \triangleright (\phi'; \Gamma)}{c[a_1] \dots [a_n](p') \downarrow \delta(j) \triangleright (a_1 : \gamma_1, \dots, a_n : \gamma_n, i \doteq j, \phi'; \Gamma)} \text{ (pat-cons-w)}$$

where $\tau = \delta(j)$, $\phi = a_1 : \gamma_1, \dots, a_n : \gamma_n, i \doteq j, \phi'$. Assume that v matches p . Then v is of form $c[i_1] \dots [i_n](v')$, v' matches p' and $\cdot_{\text{ind}} \vdash i_k : \gamma_k$ are derivable for $1 \leq k \leq n$. Since $c[i_1] \dots [i_n](v')$ is of type $\delta(i[\theta])$ for $\theta = [a_1 \mapsto i_1, \dots, a_n \mapsto i_n]$, $\cdot_{\text{ind}} \models i[\theta] \doteq j$ is satisfiable. This leads to the satisfiability of $\phi'[\theta] \models \perp$ since

$\phi \models \perp$ is satisfiable. Notice that we can also derive $\cdot_{\text{ind}}; \cdot \vdash v' : \tau'[\theta]$ and $p' \downarrow \tau'[\theta] \triangleright (\phi'[\theta], \Gamma[\theta])$. By induction hypothesis, v' does not match p' . This yields a contradiction, and therefore v does not match p .

All other cases can be treated similarly.

Dynamic Semantics The operational semantics of $\text{ML}_0^\Pi(C)$ can be given as usual in the style of natural semantics [9]. Again we omit the standard evaluation rules, which can be found in [20].

We use $e \hookrightarrow_d v$ to mean that e reduces to a value v in this semantics. These evaluation rules are only needed for proving the correctness of our approach. The following type-preservation theorem is also needed for this purpose.

Theorem 1. (*Type preservation in $\text{ML}_0^\Pi(C)$*) *Given e, v in $\text{ML}_0^\Pi(C)$ such that $e \hookrightarrow_d v$ is derivable. If $\cdot_{\text{ind}}; \cdot \vdash e : \tau$ is derivable, then $\cdot_{\text{ind}}; \cdot \vdash v : \tau$ is derivable.*

Proof. Please see Section 4.1.2 in [20] for details.

Notice that there is some nondeterminism associated with the rule for evaluating a case statement. If more than one matching clauses can match the value, there is no order to determine which one should be chosen. This is different from the deterministic strategy adopted in ML, which always chooses the first one which matches. We shall come back to this point later.

2.3 Operational Equivalence

In order to prove the correctness of our approach to dead code elimination, we must show that this approach does not alter the semantics of a program. We introduce the operational equivalence relation \cong as follows for this purpose.

Definition 1. *We present the definition of contexts as follows.*

$$\begin{array}{ll}
 \text{(matching contexts)} & C_m ::= | (p \Rightarrow C \mid ms) \mid (p \Rightarrow e \mid C_m) \\
 \text{(contexts)} & C ::= [] \mid \langle C, e \rangle \mid \langle e, C \rangle \mid c(C) \mid \mathbf{lam} \ x : \tau. C \mid C(e) \mid e(C) \\
 & \quad \mid \mathbf{case} \ C \ \mathbf{of} \ ms \mid \mathbf{case} \ e \ \mathbf{of} \ C_m \mid \mathbf{fix} \ f : \tau. C \\
 & \quad \mid \mathbf{let} \ x = C \ \mathbf{in} \ e \ \mathbf{end} \mid \mathbf{let} \ x = e \ \mathbf{in} \ C \ \mathbf{end}
 \end{array}$$

Given a context C and an expression e , $C[e]$ stands for the expression formulated by replacing with e the hole $[]$ in C . We emphasize that *this replacement is variable capturing*.

Definition 2. *Given two expression e_1 and e_2 in $\text{ML}_0^\Pi(C)$, e_1 is operationally equivalent to e_2 if the following holds.*

- *Given any context C such that $\cdot_{\text{ind}}; \cdot \vdash C[e_i] : \mathbf{1}$ are derivable for $i = 1, 2$, $C[e_1] \hookrightarrow_d \langle \rangle$ is derivable if and only if $C[e_2] \hookrightarrow_d \langle \rangle$ is.*

We write $e_1 \cong e_2$ if e_1 is operationally equivalent to e_2 .

Clearly \cong is an equivalence relation. Our aim is to prove that if a program e is transformed into \underline{e} after dead code elimination, the $e \cong \underline{e}$. In other words, we intend to show that dead code elimination does not alter the operational semantics of a program.

3 Dead Code Elimination

We now go through an example to show how dead code elimination is performed on a program. This approach is then formalized and proven correct. This constitutes the main technical contribution of the paper.

3.1 An Example

Let us declare the function `zip_safe` in DML as follows.

```
fun zip_safe(intNil, intNil) = intPairNil
  | zip_safe(intCons(x,xs), intCons(y,ys)) =
    intPairCons((x, y), zip_safe(xs, ys))
  | zip_safe _ = raise ZipException
where zip_safe <| {n:nat} intList(n)*intList(n)->intPairList(n)
```

This declaration is then elaborated (after type-checking) to the following expression in $ML_0^n(C)$ (we assume that $raise(ZipException)$ is a legal expression). Notice that the third matching clause in the above definition is transformed into two *non-overlapping* matching clauses. This is necessary since pattern matching is done sequentially in ML, and therefore the value must match either pattern $(intNil, intCons(_, _))$ or $(intCons(_, _), intNil)$ if the third clause is chosen. The approach to performing such a transform is standard and therefore omitted.

```
fix zip :  $\Pi n : nat. intList(n) * intList(n) \rightarrow intPairList(n).$ 
 $\lambda n : nat. \mathbf{lam} \ l : intList(n) * intList(n).$ 
  case l of (nil, nil)  $\Rightarrow$  nil
    | (cons[a](x, xs), cons[b](y, ys)  $\Rightarrow$  cons[a]((x, y), zip[a](xs, ys))
    | (nil, cons[b](y, ys))  $\Rightarrow$  raise(ZipException)
    | (cons[a](x, xs), nil)  $\Rightarrow$  raise(ZipException)
```

If the third clause in the above expression can be chosen, then $(nil, cons[b](y, ys))$ must match a value of type $intList(n) * intList(n)$ for some natural number n . Checking $(nil, cons[b](y, ys))$ against $intList(n) * intList(n)$, we derive the following for $\phi = (0 \doteq n, b : nat, b + 1 \doteq n)$.

$$(nil, cons[b](y, ys)) \downarrow intList(n) * intList(n) \triangleright (\phi; y : int, ys : intList(b))$$

Notice that ϕ is inconsistent since $\phi \models \perp$ is satisfiable. Therefore, we know by Lemma 1 that the third clause is unreachable. Similarly, the fourth clause is also unreachable. We can thus eliminate the dead code when compiling the program.

3.2 Formalization

While the above informal presentation of dead code elimination is intuitive, we have yet to demonstrate why this approach is correct and how it can be

implemented. For this purpose, we formalize the approach with derivation rules. A judgment of form $\phi; \Gamma \vdash e : \tau \gg \underline{e}$ means that an expression e of type τ under $\phi; \Gamma$ transforms into the expression \underline{e} through dead code elimination. The rules for deriving such a judgment are presented in Figure 4. Notice that the rule **(elim-mat-dead)** is the only rule which eliminates dead code.

$$\begin{array}{c}
\frac{\Gamma(x) = \tau}{\phi; \Gamma \vdash x : \tau \gg x} \text{ (elim-var)} \\
\\
\frac{S(c) = \Pi \vec{a} : \vec{\gamma}. \delta(i)}{\phi; \Gamma \vdash c[\vec{i}] : \delta(i[\vec{a} \mapsto \vec{i}]) \gg c[\vec{i}]} \text{ (elim-cons-wo)} \\
\\
\frac{S(c) = \Pi \vec{a} : \vec{\gamma}. \tau \Rightarrow \delta(i) \quad \phi; \Gamma \vdash e : \tau[\vec{a} \mapsto \vec{i}] \gg \underline{e}}{\phi; \Gamma \vdash c[\vec{i}](e) : \delta(i[\vec{a} \mapsto \vec{i}]) \gg c[\vec{i}](\underline{e})} \text{ (elim-cons-w)} \\
\\
\frac{}{\phi; \Gamma \vdash \langle \rangle : \mathbf{1} \gg \langle \rangle} \text{ (elim-unit)} \\
\\
\frac{\phi; \Gamma \vdash e_1 : \tau_1 \gg \underline{e_1} \quad \phi; \Gamma \vdash e_2 : \tau_2 \gg \underline{e_2}}{\phi; \Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 * \tau_2 \gg \langle \underline{e_1}, \underline{e_2} \rangle} \text{ (elim-prod)} \\
\\
\frac{\phi \vdash \Gamma[\text{ctx}] \quad \phi \vdash \tau_1 \Rightarrow \tau_2 : *}{\phi; \Gamma \vdash \cdot_{\text{ms}} : \tau_1 \Rightarrow \tau_2 \gg \cdot_{\text{ms}}} \text{ (elim-mat-empty)} \\
\\
\frac{p \downarrow \tau_1 \triangleright (\phi'; \Gamma') \quad \phi, \phi'; \Gamma, \Gamma' \vdash e : \tau_2 \gg \underline{e} \quad \phi; \Gamma \vdash ms : \tau_1 \Rightarrow \tau_2 \gg \underline{ms}}{\phi; \Gamma \vdash (p \Rightarrow e \mid ms) : \tau_1 \Rightarrow \tau_2 \gg (p \Rightarrow \underline{e} \mid \underline{ms})} \text{ (elim-mat)} \\
\\
\frac{p \downarrow \tau_1 \triangleright (\phi'; \Gamma') \quad \phi, \phi' \models \perp \quad \phi; \Gamma \vdash ms : \tau_1 \Rightarrow \tau_2 \gg \underline{ms}}{\phi; \Gamma \vdash (p \Rightarrow e \mid ms) : \tau_1 \Rightarrow \tau_2 \gg \underline{ms}} \text{ (elim-mat-dead)} \\
\\
\frac{\phi; \Gamma \vdash e : \tau_1 \gg \underline{e} \quad \phi; \Gamma \vdash ms : \tau_1 \Rightarrow \tau_2 \gg \underline{ms}}{\phi; \Gamma \vdash \text{case } e \text{ of } ms : \tau_2 \gg \text{case } \underline{e} \text{ of } \underline{ms}} \text{ (elim-case)} \\
\\
\frac{\phi, a : \gamma; \Gamma \vdash e : \tau \gg \underline{e}}{\phi; \Gamma \vdash \lambda a : \gamma. e : (\Pi a : \gamma. \tau) \gg \lambda a : \gamma. \underline{e}} \text{ (elim-ilam)} \\
\\
\frac{\phi; \Gamma \vdash e : \Pi a : \gamma. \tau \gg \underline{e} \quad \phi \vdash i : \gamma}{\phi; \Gamma \vdash e[i] : \tau[a \mapsto i] \gg \underline{e}[i]} \text{ (elim-iapp)} \\
\\
\frac{}{\phi; \Gamma, x : \tau_1 \vdash e : \tau_2 \gg \underline{e}} \\
\\
\frac{}{\phi; \Gamma \vdash \text{lam } x : \tau_1. e : \tau_1 \rightarrow \tau_2 \gg \text{lam } x : \tau_1. \underline{e}} \\
\\
\frac{}{\phi; \Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \gg \underline{e_1} \quad \phi; \Gamma \vdash e_2 : \tau_2 \gg \underline{e_2}} \\
\\
\frac{}{\phi; \Gamma \vdash e_1(e_2) : \tau_2 \gg \underline{e_1}(\underline{e_2})} \\
\\
\frac{\phi; \Gamma \vdash e_1 : \tau_1 \gg \underline{e_1} \quad \phi; \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \gg \underline{e_2}}{\phi; \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \text{ end} : \tau_2 \gg \text{let } x = \underline{e_1} \text{ in } \underline{e_2} \text{ end}} \text{ (elim-let)} \\
\\
\frac{\phi; \Gamma, f : \tau \vdash u : \tau \gg \underline{u}}{\phi; \Gamma \vdash (\text{fix } f : \tau. u) : \tau \gg \text{fix } f : \tau. \underline{u}} \text{ (elim-fix)}
\end{array}$$

Fig. 4. The rules for dead code elimination

Proposition 1. *We have the following.*

1. *If $\phi; \Gamma \vdash e : \tau$ is derivable, then $\phi; \Gamma \vdash e : \tau \gg e$ is also derivable.*
2. *if $\phi; \Gamma \vdash e : \tau \gg \underline{e}$ and $\cdot_{\text{ind}}; \cdot \vdash C[e] : \mathbf{1}$ are derivable, then $\cdot_{\text{ind}}; \cdot \vdash C[e] : \mathbf{1} \gg C[\underline{e}]$ is also derivable.*
3. *If $\phi; \Gamma \vdash e : \tau \gg \underline{e}$ is derivable, then both $\phi; \Gamma \vdash e : \tau$ and $\phi; \Gamma \vdash \underline{e} : \tau$ are derivable.*

Proof. (1) and (2) are straightforward, and (3) follows from structural induction on the derivation of $\phi; \Gamma \vdash e : \tau \gg \underline{e}$.

Proposition 1 (1) simply means that we can always choose not to eliminate any code, and (2) means that dead code elimination is independent of context, and (3) means that dead code elimination is type-preserving.

Lemma 2. (*Substitution*)

1. *If both $\phi; \Gamma, x : \tau_2 \vdash e_1 : \tau_1 \gg \underline{e}_1$ and $\phi; \Gamma \vdash e_2 : \tau_2 \gg \underline{e}_2$ are derivable, then so is $\phi; \Gamma \vdash e_1[x \mapsto e_2] : \tau_1 \gg \underline{e}_1[x \mapsto \underline{e}_2]$.*
2. *If both $\phi, a : \gamma; \Gamma \vdash e : \tau \gg \underline{e}$ and $\phi \vdash i : \gamma$ are derivable, then so is $\phi; \Gamma[a \mapsto i] \vdash e[a \mapsto i] : \tau[a \mapsto i] \gg \underline{e}[a \mapsto i]$.*

Proof. (1) and (2) follow from structural induction on the derivations of $\phi; \Gamma, x : \tau_2 \vdash e_1 : \tau_1 \gg \underline{e}_1$ and $\phi, a : \gamma; \Gamma \vdash e : \tau \gg \underline{e}$, respectively.

Lemma 3. *Assume that $\cdot_{\text{ind}}; \cdot \vdash e : \tau \gg \underline{e}$ is derivable.*

1. *If $e \hookrightarrow_d v$ is derivable, then $\underline{e} \hookrightarrow_d \underline{v}$ is derivable for some \underline{v} such that $\cdot_{\text{ind}}; \cdot \vdash v : \tau \gg \underline{v}$ is also derivable.*
2. *If $\underline{e} \hookrightarrow_d \underline{v}$ is derivable, then $e \hookrightarrow_d v$ is derivable for some v such that $\cdot_{\text{ind}}; \cdot \vdash v : \tau \gg \underline{v}$ is also derivable.*

Proof. (1) and (2) follow from structural induction on the derivations of $e \hookrightarrow_d v$ and $\underline{e} \hookrightarrow_d \underline{v}$, respectively, using both Lemma 1 and Lemma 2.

Theorem 2. *If $\phi; \Gamma \vdash e : \tau \gg \underline{e}$ is derivable, then $e \cong \underline{e}$ holds.*

Proof. Let C be a context such that both $\cdot_{\text{ind}}; \cdot \vdash C[e] : \mathbf{1}$ and $C[e] \hookrightarrow_d \langle \rangle$ are derivable. By Proposition 1, $\cdot_{\text{ind}}; \cdot \vdash C[e] : \mathbf{1} \gg C[\underline{e}]$ is derivable. Hence, by Lemma 3 (1), $C[\underline{e}] \hookrightarrow_d \underline{v}$ is derivable for some \underline{v} such that $\cdot_{\text{ind}}; \cdot \vdash \langle \rangle : \mathbf{1} \gg \underline{v}$. This implies that \underline{v} is $\langle \rangle$. Similarly, by Lemma 3 (2), we can prove that $C[e] \hookrightarrow_d \langle \rangle$ is derivable for every context C such that both $\cdot_{\text{ind}}; \cdot \vdash C[\underline{e}] : \mathbf{1}$ and $C[\underline{e}] \hookrightarrow_d \langle \rangle$ are derivable. Therefore, $e \cong \underline{e}$ holds.

4 Examples

We present some realistic examples in this section to demonstrate the effect of dead code elimination through dependent types. Note that polymorphism is allowed in this section.

4.1 The Nth Function

When applied to (l, n) , the following `nth` function returns the n th element in the list l if n is less than the length of l and raises the `Subscript` exception otherwise. This function is frequently called in functional programming, where the use of lists is pervasive.

```
fun nth(nil, _) = raise Subscript
  | nth(x::xs, n) = if n = 0 then x else nth(xs, n-1)
```

If we assign `nth` the following type, that is, we restrict the application of `nth` to pairs (l, n) such that n is always less than the length of l ,

```
{len:nat}{index:nat | index < len} 'a list(len) * int(index) -> 'a
```

then the first matching clause in the definition of `nth` is unreachable, and therefore can be safely eliminated. Note that we have refined the built-in type `int` into infinitely many singleton types such that `int(n)` contains *only* n for each integer n . Let us call this version `nth_safe`, and we have measured that `nth_safe` is about 25% faster than `nth` on a Sparc 20 station running SML/NJ version 110. The use of a similar idea to eliminate array bound checks can be found in [17].

4.2 An Evaluator for the Call-by-Value λ -Calculus

The code in Figure 5 implements an evaluator for the pure call-by-value λ -calculus. We use de Bruijn's notation to represent λ -expressions. For example, $\lambda x. \lambda y. y(x)$ is represented as `Lam(Lam(App(One, Shift(One))))`. We then refine the datatype `lambda_exp` into infinitely many types `lambda_exp(n)`. For each natural numbers n , `lambda_exp(n)` roughly stands for the type of all λ -terms in which there are *at most* n free variables.

If a value of type `lambda_exp(n) * closure list(n)` for some n matches the last clause in the definition of `cbv` then it matches either pattern `(One, nil)` or pattern `(Shift _, nil)`. In either case, a contradiction is reached since a value which matches either `One` or `Shift _` can not be of type `lambda_exp(0)` but `nil` is of type `closure list(0)`. Therefore, the last clause can be safely eliminated.

The programmer knows that the last clause is unreachable. After this is mechanically verified, the programmer gains confidence in the above implementation. On the other hand, if the last clause could not be safely eliminated, it would have been an indication of some program errors in the implementation.

4.3 Other Examples

So far all the presented examples involve the use of lists, but this is not necessary. We also have examples involving other data structures such as trees. For instance, the reader can find in [19] a red/black tree implementation containing unreachable matching clauses which can be eliminated in the same manner. In general, unreachable matching clauses are abundant in practice, of which many can be eliminated with our approach.

```

datatype lambda_exp =
  One | Shift of lambda_exp |
  Lam of lambda_exp | App of lambda_exp * lambda_exp

datatype closure = Closure of lambda_exp * closure list

typeref lambda_exp of nat
with One <| {n:nat} lambda_exp(n+1)
  | Shift <| {n:nat} lambda_exp(n) -> lambda_exp(n+1)
  | Lam <| {n:nat} lambda_exp(n+1) -> lambda_exp(n)
  | App <| {n:nat} lambda_exp(n) * lambda_exp(n) -> lambda_exp(n)
  | Closure <| {n:nat} lambda_exp(n) * closure list(n) -> closure

exception Unreachable

fun callbyvalue(exp) = let
  fun cbv(One, clo::_) = clo
    | cbv(Shift(exp), _::env) = cbv(exp, env)
    | cbv(exp as Lam _, env) = Closure(exp, env)
    | cbv(App(exp1, exp2), env) = let
        val Closure(Lam(body), env1) = cbv(exp1, env)
        and clo = cbv(exp2, env)
      in cbv(body, clo::env1) end
    | cbv _ = raise Unreachable (* this can be safely eliminated *)
  where cbv <| {n:nat} lambda_exp(n) * closure list(n) -> closure
in
  cbv(exp, nil)
end
where callbyvalue <| lambda_exp(0) -> closure
(* Note: callbyvalue can only apply to CLOSED lambda expressions *)

```

Fig. 5. An evaluator for the call-by-value λ -calculus

5 Related Work and Conclusion

It is beyond reasonable hope to mention even a moderate amount of research related or similar to dead code or dead computation elimination because of the vastness of the field. The reader can find further references in [8, 1, 7, 13, 10, 14, 11, 15]. Our approach to dead code elimination differs significantly from the previous approaches in several aspects.

We have adopted a type-based approach while most of the previous approaches are based on flow analysis. This gives us a great advantage when the issue of crossing module boundaries is concerned. For instance, after assigning the zip function the type

$$\{n:nat\} \text{ 'a list}(n) * \text{ 'b list}(n) \rightarrow (\text{ 'a} * \text{ 'b}) \text{ list}(n)$$

and eliminating the dead code, we can use this function *anywhere* as long as type-checking is passed. On the other hand, an approach based on flow analysis usually analyzes an instance of a function call and check whether there is dead code associated with this *particular* function call. One may argue that our approach must be supported by a dependent type system while an approach based on flow analysis need not. However, there would be no dead code in the `zip` function if we had not assigned it the above dependent type. It is the use of a dependent type system that enables us to exploit opportunities which do not exist otherwise.

Also we are primarily concerned with program error detection while most of the previous approaches were mainly designed for compiler optimization, which is only our secondary goal. Again, this is largely due to our adoption of a type-based approach.

We emphasize that our approach must rely on the type annotations supplied by the programmer in order to detect redundant matching clauses. It seems exceedingly difficult at this moment to find a procedure which can synthesize type annotations automatically. For instance, without the type annotation in the `zip_safe` example, it is unclear whether the programmer intends to apply the function to a pair lists of unequal lengths, and therefore unclear whether the last matching clause is redundant.

Our approach is most closely related to the research on refinement types [4, 2], which also aims for assigning programs more accurate types. However, the restricted form of dependent types in DML allows the programmer to form types which are not captured by the regular tree grammar [5], e.g., the type of all pairs of lists of equal length, but this is beyond the reach of refinement types. The price we pay is the loss of principal types, which may consequently lead to a more involved type-checking algorithm.

We have experimented our approach to dead code elimination in a prototype implementation of a type-checker for DML. We plan to incorporate this approach into the compiler for DML which we are building on top of Caml-light. Clearly, our approach can also be readily adapted to detecting uncaught exceptions [21], and we expect it to work well in this direction when combined with the approach in [3]. We shall report the work in the future.

References

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers*. Addison-Wesley, Reading, Massachusetts, 1986.
- [2] Rowan Davies. Practical refinement-type checking. Thesis Proposal, November 1997.
- [3] Manuel Fähndrich and Alexander Aiken. Program analysis using mixed term and set constraints. In *Proceedings of the 4th International Static Analysis Symposium*, September 1997.
- [4] Tim Freeman and Frank Pfenning. Refinement types for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 268–277, Toronto, Ontario, 1991.
- [5] F. Gecseg and M Steinb. *Tree automata*. Akademiai Kiado, 1991.

- [6] Paul Hudak, S. L. Peyton Jones, and Philip Wadler. Report on the programming language Haskell, a non-strict purely-functional programming language, Version 1.2. *SIGPLAN Notices*, 27(5), May 1992.
- [7] John Hughes. Compile-time analysis of functional programs. In D. Turner, editor, *Research Topics in Functional Programming*, pages 117–153. Addison-Wesley, 1990.
- [8] N. Jones and S. Muchnick. Flow analysis and optimization of lisp-like structures. In *Conference Record of 6th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 244–256, January 1979.
- [9] Gilles Kahn. Natural semantics. In *Proceedings of the Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer-Verlag LNCS 247, 1987.
- [10] J. Knoop, O. Rüthing, and B. Steffen. Partial dead code elimination. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Implementation and Design*, pages 147 – 158, June 1994.
- [11] Y. Liu and D. Stoller. Dead code elimination using program-based regular tree grammars. Available at <http://ftp.cs.indiana.edu/pub/liu/ElimDeadRec-TR97.ps.Z>.
- [12] Robin Milner, Mads Tofte, Robert W. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1997.
- [13] Heintze Nevin. *Set-based Program Analysis of ML programs*. Ph. D dissertation, Carnegie Mellon University, 1992.
- [14] F. Tip. A survey of program slicing. *Journal of Programming Languages*, 3(3):121–189, 1995.
- [15] P. Wadler and J. Hughes. Projections for strictness analysis. In *Proceedings of the 3rd International Conference on Functional Programming and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 385 – 407. Springer-Verlag, 1987.
- [16] Pierre Weis and Xavier Leroy. *Le langage Caml*. InterEditions, Paris, 1993.
- [17] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, June 1998.
- [18] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, January 1999.
- [19] Hongwei Xi. Some examples of DML programming. Available at <http://www.cs.cmu.edu/~hwxi/DML/examples/>, November 1997.
- [20] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998. pp. viii+189. Forthcoming. The current version is available as <http://www.cs.cmu.edu/~hwxi/DML/thesis.ps>.
- [21] K. Yi and S. Ryu. Towards a cost-effective estimation of uncaught exceptions in standard ml programs. In *Proceedings of the 4th International Static Analysis Symposium*, September 1997.

Multiple Specialization of WAM Code

Michel Ferreira* and Luís Damas

DCC-FC & LIACC, University of Porto,
Rua do Campo Alegre, 823 - 4150 Porto, Portugal
`{michel,luis}@ncc.up.pt`

Abstract. Program specialization is normally supported by global analysis of the program. Compilers use the information deduced to generate more efficient, specialized implementations of the program. This specialization can be *single* or *multiple*, depending if each procedure of the program is specialized into one or more versions. We present a Prolog compiler that does *multiple specialization*, using an algorithm that works over the WAM code, deducing the reachable procedure activations based on local analysis. The idea is to use the analysis that is done in the Prolog to WAM compilation, and that generates local specialized instructions, and to propagate this information through the entire program. The creation of multiple versions of predicates allows the propagation to be interprocedural, and to achieve global program specialization. Results from an implementation generating C code as target language are presented, showing a performance comparable to native code Prolog compilers.

Keywords: Prolog Compilation, Code Specialization, Program Analysis.

1 Introduction

The efficiency of a Prolog compiler is normally associated with two aspects: a program analyzer and the generation of native code. Parma [14] and Aquarius [15] are two of the most efficient Prolog compilers and both generate native code and have a program analysis phase.

The analysis phase is normally done by applying a technique named *abstract interpretation* [4] to the source program. This analysis generates information about modes and types of arguments [5, 2], variable aliasing [9], determinism [11], etc. This information is then used to generate specialized code, normally in the form of some low level abstract machine, like the BAM (Berkeley Abstract Machine) [15] in Aquarius. The WAM (Warren Abstract Machine) [17] is much too high level to use the information of the analysis.

The code specialization results from knowing that the activation patterns from procedure calls have certain properties, like, for instance, having dereferenced arguments, non-instantiated arguments or of integer type. A new specialized version of the procedure is then generated, making use of the properties

* The first author is thankful to PRAXIS and Fundação para a Ciência e Tecnologia for their financial support.

deduced. If all the activation patterns of a certain procedure that occur in a program are similar, then the specialized code of that procedure can be importantly optimized. If however there are very distinct (or unknown) activation patterns for a procedure, then the optimizations are limited. An example of unknown activation patterns occurs on modular programming, when not all modules are available for analysis. This kind of code specialization is known as *single specialization*, because it is generated just one specialized version for each procedure. Parma and Aquarius are, basically, single specializations compilers.

Winsborough proposed in [18] another kind of specialization, where a different specialized version was generated for each deduced reachable activation pattern. This is known as *multiple specialization*. The way to deduce the kind of activation patterns that were reachable during execution was also by using abstract interpretation over the source program.

In this paper we present an algorithm to obtain multiple specialization that is not based on abstract interpretation, showing how a local analysis on the WAM instructions of a predicate can be used to deduce the activation patterns of the calls and how the creation of multiple versions of predicates allows the propagation of such information to be inter-procedural, achieving global program specialization.

The remainder of this paper is organized as follows: in the following section we discuss some techniques of program analysis, as the tool to derive the information needed for program specialization; next we present our strategy to obtain specialized code, showing how we can simplify this analysis; we then discuss the implementation of the WAM instructions generated for the multiple specialized predicates; next we present evaluation results of the performance of our system and we end with some conclusions.

2 Abstract Interpretation and Abstract Compilation

To obtain code specialization, single or multiple, the framework of abstract interpretation has been widely used, as the tool to global data-flow analysis of a programming language based on an unified model [4], in order to extract information from programs, like mode information, type or variable aliasing.

The idea is to execute a program over a finite *abstract domain* rather than the possibly infinite *concrete domain*. Elements and operators from the *abstract domain* map a class of elements and operators of the *concrete domain*. The abstract interpretation proceeds until a fix-point of the program properties is reached. New definitions of unification and variable substitutions are needed in this abstract execution.

The major problem with abstract interpretation is that it is a very time consuming process. Also, the complexity of real programs makes obtaining precise data rather difficult, deriving naive results that do not pay back the time spent in the analysis phase.

Another method of analysis has been proposed that tries to overcome the large time-consumption of abstract interpretation, and is known as *abstract com-*

pilation [8]. It is based on the simple idea that instead of interpreting the source program it should be compiled into a program which computes the desired analysis when it is executed. The removal of abstraction to the analysis process should result in an important improvement on execution speed.

An implementation of the *abstract compilation* technique in the context of logic programs analysis was done by Tan and Lin in [13]. However, their approach is not a pure abstract compilation implementation, because there is no notion of analysis program, as what they do is an *abstract interpretation of the WAM compiled code*.

This approach is however very interesting and has a similar motivation of the idea we describe in this paper: in the WAM code there are important specializations, for instance in specialized unification subroutines, defined for each type of term (`_variable`, `_value`, `_const`, `_list`, `_struct`), that in [13] are used to simplify the analysis and that we use with a propagation technique to generate multi-specialized code.

3 Unfolding WAM Code

The multi-directionality of Prolog is well appreciated by the programmer, but is a problem to the compiler designer. When no modes declarations are given or deduced, the code that is generated for a predicate will answer calls from different “directions”, handicapping the efficiency of the code with several run-time tests, to implement distinct cases in one general procedure.

Recursive predicates are extremely used in Prolog, and recursive calls show how those run-time tests are redundantly executed. If a specialized predicate was created for each basic direction of execution then those run-time tests could be importantly minimized.

In Figure 1 we present some specialized versions of the `append/3` predicate. Those specialized versions are the versions that our algorithm of multiple specialization generates for the `append/3` predicate. In the next sections we will show how they are generated.

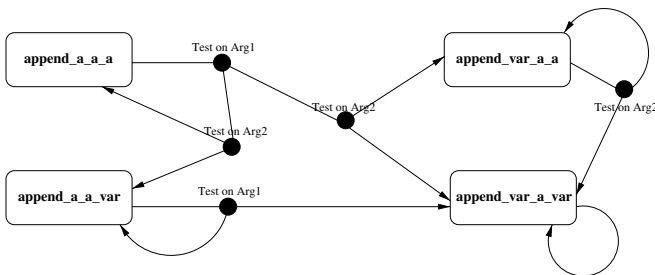


Fig. 1. Call graph on unfolded append program

The `append_a_a1` predicate is the general `append/3` predicate. In a single version and non-specialized implementation this will be the only predicate called. Two important run-time tests are executed, the first one on argument 1, corresponding to the `get_list(1)` instruction and the second one on argument 3, corresponding to the `get_list(3)` instruction. The WAM code for the recursive clause of the `append/3` predicate is presented in Figure 2.

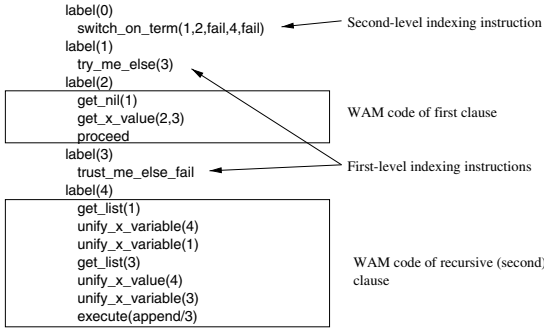


Fig. 2. WAM code of `append/3`

If the result from these run-time tests was propagated to the calling instructions, leading the calls to specialized versions as is represented in the graph on Figure 1 then in subsequent calls those run-time tests could be partially or totally eliminated.

3.1 Propagation of Information for Inter-procedural Specialization

The `get_list` instruction is a specialization of the `get_structure` instruction, where the functor is `./2`. This allows the implementation of such unification to be simplified. The kind of specialization present in WAM instructions is local to instructions, or in the best is intra-procedural, but an inter-procedural specialization is non-existent. To obtain inter-procedural specialization analysis is needed. Our idea is to propagate the local specialization information of WAM instructions through the entire code of the program, avoiding a global analysis phase. In calling instructions the information propagated until the call point will describe the activation pattern of that call, and will lead to the creation of the correspondent specialized version, allowing the propagation of information to be inter-procedural. A similar work that also tries to relate procedure calls directly with heads unification is present in the VAM [10], where two program pointers are used, one to the calling goal and one to the head of the unify-able clause.

¹ `a` stands for type *any*, describing the type of the first, second and third arguments, respectively.

When an important run-time test is done, like the `get_list` in `append/3`, and information about the argument tested is not present, the code will be unfolded to propagate the succeeding possible types through the distinct branches. This happens in figure 1 where tests on arguments which have type *any* cause code to be split in two, with different `execute` instructions that call different predicates.

The process of collecting information from WAM instructions and propagating it is very simple and we will describe it using the `append/3` example.

The information we need to propagate and maintain is the following:

- type of temporary and global variables
- mode of structure register
- possible sharing of local and global variables

Figure 2 presents the WAM code of the `append/3` predicate, with labels and indexing instructions. To a detailed explanation of WAM instructions see 1.

In this example the first run-time test is done by the second-level indexing instruction `switch_on_term(1,2,fail,4,fail)`. It tests the tag of argument 1 and jumps to label 1 if it is *var*, to label 2 if it is *const* or to label 4 if it is *list*. If it is *integer* or *struct* execution fails.

As nothing is known about the type of argument 1 (we are applying our algorithm to the general `append/3` WAM code), we will propagate the succeeding possible types (*var*, *const* or *list*) through the correspondent distinct branches. The propagation of different information for each distinct branch leads to different implementations of the WAM instructions, and no longer makes possible to interlace the code of the various cases, as is done in Figure 2. The code of clauses now has to be repeated to the *var*, *const* and *list* cases.

To distinguish the implementation of WAM instructions in each of these cases, based on the information propagated, we complement the instructions with the relevant information to their simplification. A `get_list(1)` instruction where it has been propagated that register argument 1 has type *var* becomes `get_list_var(1)`. This specialized instruction propagates the unification mode to be *write*, which will complement the subsequent `unify` instructions.

Some important run-time tests, like the `get_list(3)` in our example, where no information is present about the argument, will be replaced by an instruction which splits the code in distinct branches and jumps to the correspondent branch accordingly to the test made. A `switch_on_list(3,3,4)` will replace a `get_list(3)` instruction where argument 3 has type *any*, testing the type of this argument and jumping to label 3 if it has type *list* or to label 4 if it has type *var*.

The calling instructions (`execute` in our example) are complemented with the propagated type of their arguments, calling a correspondent specialized predicate that will be created.

In Figure 3 we present the WAM unfolded code of `append_a_a_a/3` after applying our algorithm of specialization.

The difference between the code of Figure 2 and the code of Figure 3 is that in this last code we have propagated the information of the run-time tests

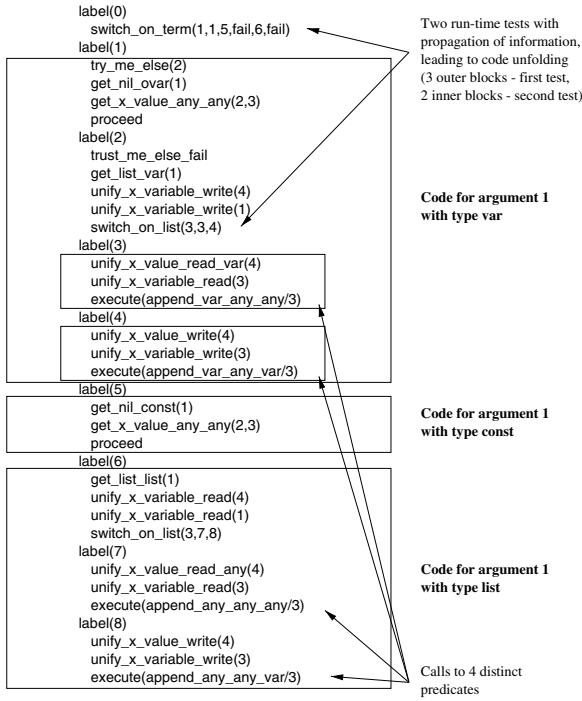


Fig. 3. The unfolded WAM code for `append_a_a_a/3`

that are made, causing blocks of code to be repeated because of implementation differences, and obtaining 4 distinct predicate calls.

3.2 Creation of Specialized Predicates

The inter-procedural specialization is obtained through the propagation of the activation pattern of the calling goal into the predicate which will handle the call. In our algorithm this is automatically achieved, because each activation pattern of the calling goal that is deduced through the process of propagation, will lead to the creation of a predicate that is specialized in that activation pattern.

For each distinct reachable activation pattern it will be created a specialized predicate. In our `append/3` example there are four distinct activation patterns in the calls to `append/3`: `append_a_a_a`, `append_var_a_a`, `append_var_a_var` and `append_a_a_var`.

Whenever a calling instruction is found when applying our specialization algorithm, the new predicate will be created and the algorithm of specialization will be applied to it. The information propagated through the arguments types will be used to simplify the codification. In the `append_var_a_a`, for instance, no `switch_on_term` instruction has to be generated, as the tag of argument 1 is

already known to be *var*. If when applying the algorithm to the created predicates, new activation patterns are derived, they will also give rise to predicates and the process will also be applied on them, until no more distinct activation patterns are derived.

Note that splitting a predicate into several predicates makes execution more deterministic. Clause selection is done based on the unification of activation patterns and clauses heads. Indexation is also more powerful as it takes advantage from the knowledge of the types of the arguments, allowing to index on the argument that creates less choice-points⁴.

Figure 4 shows the code for the specialized versions generated.

append_var_a_a	append_var_a_var	append_a_a_var
label(0) try_me_else(1) get_nil_ovar(1) get_x_value_any_any(2,3) proceed label(1) trust_me_else_fail get_list_var(1) unify_x_variable_write(4) unify_x_variable_write(1) switch_on_list(3,2,3) label(2) unify_x_value_read_var(4) unify_x_variable_read(3) execute(append_var_a_a/3) label(3) unify_x_value_write(4) unify_x_variable_write(3) execute(append_var_a_var/3)	label(0) try_me_else(1) get_nil_ovar(1) get_x_value_any_ovar(2,3) proceed label(1) trust_me_else_fail get_list_var(1) unify_x_variable_write(4) unify_x_variable_write(1) get_list_var(3) unify_x_value_write(4) unify_x_variable_write(3) execute(append_var_a_var/3)	label(0) switch_on_term(1,1,3,fail,4,fail) label(1) try_me_else(2) get_nil_ovar(1) get_x_value_any_ovar(2,3) proceed label(2) trust_me_else_fail get_list_var(1) unify_x_variable_write(4) unify_x_variable_write(1) get_list_var(3) unify_x_value_write(4) unify_x_variable_write(3) execute(append_var_a_var/3) label(3) get_nil_const(1) get_x_value_any_var(2,3) proceed label(4) get_list_list(1) unify_x_variable_read(4) unify_x_variable_read(1) get_list_var(3) unify_x_value_write(4) unify_x_variable_write(3) execute(append_a_a_var/3)

Fig. 4. WAM code for the specialized versions generated

3.3 Some Problems with Our Algorithm

Consider the following Prolog program:

`p(X) :- q(Y), p(Y).`

`q(a).`

⁴ the first argument on the `switch_on_term` instruction of Figure 3 identifies the argument of indexing

The WAM code for this program is:

```

p/1:  allocate(1)           q/1:  get_constant(a,1)
      put_y_variable(1,1)   proceed
      call(q/1)
      put_y_value(1,1)
      deallocate
      execute(p/1)

```

Applying our process of specialization to this code, will deduce that the only activation pattern to `q/1` will be an unbound argument in register 1, allowing the `get_constant` instruction to be implemented by a store instruction. However our algorithm does not conclude anything about the activation patterns of the predicate `p/1`, although it is not difficult to see that it will always be just one, with constant `a` in argument 1.

The reason for this unconclusion is the `call` instruction. This instruction has the inconvenient effect of promoting the unbound variables present in its arguments to the type *any*. As a result an important loss of precision in the information propagated occurs. Clearly this promotion has to be done, as we do not collect and propagate exit information, and there is no way to know what the called predicate will do to unbound variables.

To avoid this loss of precision a simple analysis algorithm is used to derive the predicates argument types on exit. After a call instruction, instead of promoting unbound variables to the type *any*, we update their type based on the information about the exit types derived by the analysis algorithm.

This analysis algorithm is extremely simple, working with a bottom-up strategy, starting with facts and iteratively closing goals in clauses, until the clauses can be closed and information about the head of the clause derived.

Note that the unfolding algorithm could work without this analysis, but more specialized predicates can be created if the analysis on the exit types of predicates is performed.

Another problem with our algorithm of specialization is the code expansion. Our specialization process generates, in the worst case, a new predicate to every combination of types in the arguments. This can lead to a major code growth, that may not be supported in programs of large size. Also, a large number of predicates in a program have no advantages in being unfolded, i.e. multi-specialized, as they are not relevant to performance. It is not unusual that a 1000 lines program spends 90% of the time doing append. Our system allows the user to specify a list of the predicates to unfold, granting him with the choice of the ratio performance/code size³. If the user doesn't want to choose which predicates to unfold, the compiler chooses them, by detecting predicates which define cycles of a predefined order, being by that reason, good candidates to unfold. In the benchmark programs we have compiled this cycle strategy has given reasonable results. A more accurate strategy would be based on profiling as do some compilers of other programming languages.

³ Although this is not a monotonic ratio, due to cache problems

Some combinations of argument types give rise to a specialized code that is identical to the code of another combination of types that generalizes the one just created. In this cases we eliminate this predicate and replace the calls to it by calls to the predicate that generalizes it.

Another problem with our algorithm is that it has some limitations in the types deduced, because information is collected from WAM instructions. Our domain of types is any, var, ovar⁴, struct, list, const and integer, which basically reflects the specializations present in WAM instructions. There is no type to inform on dereferencing length as in Parma or Aquarius. This kind of information has to be deduced based on analysis. To extend the types of our domain we propose to follow the strategy described in the next subsection.

3.4 From WAM Unfolded Code Back to Prolog

Prolog compilation has improved largely since the WAM development [16], and it is, at least, at the same level of technology of imperative languages compilation. The speed of execution, however, does not follow this equality. This is due to the programming style of declarative languages, were the absence of “how to get it” concerns, leads programmers to write “what they want” with a generality that is not correct in the context of the particular program execution.

Our multiple specialization algorithm solves this problem, as it creates execution driven specialized procedures.

The Prolog code of a program can be easily reconstructed from the WAM code, and some compilers allow this to be done. It seemed interesting to us to see the Prolog programming style of a program reconstructed from our WAM unfolded code. From this source program it should not be difficult to a no-analysis compiler to achieve an efficient code.

The reconstructed Prolog code of the predicates `append_a_a_a/3` (WAM code in Figure 3) and `append_var_a_var/3` is presented in Figure 5.

The `$ovar`, `$var`, `$list` and `$const` are run-time testing instructions. In the predicate `append_var_a_var/3` these run-time tests are eliminated. Note that the unification (`=`) will have different implementations, depending on the information available about the terms to unify. When the left side is known to be `var` this unification is implemented by a store instruction. When the two terms to unify are both lists then a specialized unification will be performed.

The Prolog code obtained from this reconstruction is clearly difficult to generate by a programmer. The generation of such code by the compiler has, however, some interesting aspects. The Prolog obtained is a flat and clear code, suitable to the application of analysis algorithms, similar to the Kernel Prolog used by Aquarius analysis. Much more important is the presence of the `$ovar`, `$var`, `$list` and `$const` instructions which gives information about the type of local variables, thus simplifying analysis. Also very important is the fact that the execution path is unfolded, deriving many and more precise (from the analysis

⁴ var that needs trailing

<pre> append_a_a_a(A,B,C) :- \$ovar(A), A= [], C=B. append_a_a_a(A,B,C) :- \$var(A), A= [D E], \$list(C), C=[D F], append_var_a_a(E,B,F). append_a_a_a(A,B,C) :- \$var(A), A= [D E], \$var(C), C=[D F], append_var_a_var(E,B,F). append_a_a_a(A,B,C) :- \$const(A), A= [], C=B. </pre>	<pre> append_a_a_a(A,B,C) :- \$list(A), A= [D E], \$list(C), C=[D F], append_a_a_a(E,B,F). append_a_a_a(A,B,C) :- \$list(A), A= [D E], \$var(C), C=[D F], append_a_a_var(E,B,F). append_var_a_var(A,B,C) :- A= [], C=B. append_var_a_var(A,B,C) :- A=[D E], C=[D F], append_var_a_var(E,B,F). </pre>
---	--

Fig. 5. Prolog code reconstructed from WAM unfolded code

point of view) execution paths. Therefore we claim that running abstract interpretation algorithms over this code, will significantly improve both precision and execution time of analysis.

4 WAM Instructions Implementation

The implementation of WAM instructions has two main approaches: constructing an emulator or generating native code. The first option has a more simple implementation, and is portable to different machines. The disadvantage is poor efficiency because of the cycle *fetch*, *decode* and *execute*. The second option is clearly used when efficiency is the main issue, sacrificing portability and simplicity of implementation.

Another approach has been implemented in the **wamcc** [3] compiler. The idea is to introduce a new intermediate step between the WAM code generation and the native code. This intermediate step consists in translating the WAM code to C code. The native code generation is then obtained thanks to the C compiler.

This approach has clearly some advantages. In the end we have native code, so performance can be high. Portability is achieved as a C compiler is available to every machine. Also, it is much more simpler to generate C code than to generate native code.

The disadvantage with this approach is that the fact of having native code in the end, does not imply high performance by itself. The native code generation just exposes the bare machine, but is up to the compiler to take advantage from the low level availability.

In the introduction we referred that high performance Prolog implementation was based on two aspects: program analysis and native code generation. We have

efficiency as our main goal, and we showed in the above sections how specialization could be achieved without requiring global program analysis. We opted to generate C code in the `wamcc` style, to show that we could also eliminate native code generation and still achieve the performance of native code systems. The portability and modularity also contributed to this option. Remember that with our multiple specialization modular programming is well accepted, as an external unknown activation pattern will be handled by the always present general case, that leads subsequent calls to the correct specialized version.

Another important reason to the C generation option was the simplification of our work. We have generated C code using the `wamcc` framework, taking advantage of an intelligent control flow codification [3]. The remaining codification solutions of `wamcc` are quite conventional and high-level, and we have lost performance by reusing them, even with our more low-level complemented WAM instructions.

The C generation is an alternative to the emulation or native code generation, but to an implementation aiming maximum efficiency it is not a good option as it disallows many important low-level optimizations.

A portable assembler language, C-, with a syntax similar to C but with added low-level features, like for instance a no-alias directive, which defines the independence of two variables, has been proposed in [12], and seems like a good alternative to achieve performance without having to generate native code.

4.1 Generating C Code

The major problem to solve in coding WAM to C is the codification of the control execution. As the WAM code is flat, the execution control is done by jumps, which can be direct or indirect (the destination is the contents of some variable). The C language does not offer much to express such jumps, specially in a low level manner. Other compilers of committed choice languages addressed the problem of translating the execution control, like Janus [6] or Erlang [7]. Their methods either generated just one C function in order to be able to implement jumps as `gotos` (with different solutions to implement non-ANSI C indirect `gotos`), or implemented jumps as functions calls that could return to a supervisor function.

The `wamcc` system had the objective of translating a WAM jump by a native code jump. Because the code has to be decomposed in several functions, these jumps should reach blocks of code inside a function. The `wamcc` solution was then to insert a label at the entry of each function, thanks to `asm(...)` directives. To manipulate the addresses of those labels `wamcc` declared a prototype of an external function with the same name (address) of the label. The compiler then generates an instruction with an hole that will be filled by the linker with the knowledge of all inserted labels.

Besides the codification of the control execution, there is no relevant implementation issue in the way we generate C code. Basically WAM instructions are macro expanded to their C code implementation. In this implementation the important aspect is the simplifications allowed by the different information that

complements standard WAM instructions. In Figure 6 we present the simplified implementations of the distinct possible cases of the `get_list` instruction.

<code>get_list(A)</code>	<pre> if (!Get_List(A(a))) goto lab_fail; </pre>
<code>get_list_var(A)</code>	<pre> *(UnTag_REF(A(a)))=Tag_Value(LST,H); S=WRITE_MODE; </pre>
<code>get_list_ovar(A)</code>	<pre> TrailBind_UV(UnTag_REF(A(a)),Tag_Value(LST,H)) S=WRITE_MODE; </pre>
<code>get_list_list(A)</code>	<pre> S=(WamWord *) UnTag_LST(A(a)) + OFFSET_CAR; </pre>
<p>Get_List(A) function:</p> <pre> Bool Get_List(WamWord start_word) { WamWord word, tag, *adr; Deref(start_word,word,tag,adr); switch(tag) { case REF: TrailBind_UV(adr,Tag_Value(LST,H)) S=WRITE_MODE; return TRUE; case LST: S=(WamWord *) UnTag_LST(word) + OFFSET_CAR; return TRUE; } return FALSE; } </pre>	

Fig. 6. WAM code for the specialized versions generated

Note the number of run-time tests that are eliminated in the specialized versions. Note also how a `TrailBind` instruction is implemented by a simple store in the `get_list_var` instruction.

4.2 Specializing the Execution Model

Besides the specialization of WAM instructions, the WAM execution model can also be specialized to take advantage of the information available. For instance, we have created specialized structures for choice-points that do not need local variables. Also, argument registers known to be integers use the native integer representation, eliminating tagging and untagging operations. With this native representation we have gained almost 10% in the execution time of `tak` and `fib` benchmarks.

The multiple specialization approach makes a little harder to do this kind of model specialization, in relation with a single specialization approach, because different versions of predicates have different argument types, and not all of them may use the native representation. The solution is to tag a native representation before a generic predicate is called, and to convert to a native representation a tagged value when a specialized version is called. Inside the specialized predicate

the operations will be done over the native representation, and will be tagged before returning.

5 Performance Evaluation

We will now evaluate the performance of our system, comparing it with the `wamcc` basic system and with the Sicstus compiler, emulated and native. We also present results for the code expansion of our multiple specialization strategy, comparing the sizes of the files generated by `wamcc` and by our system (`ms-wam`).

Table 1 shows the execution times of `ms-wam` 1.0, `wamcc` 2.22, emulated Sicstus 3.0 and native Sicstus 3.0 on a set of benchmarks. Timings are in seconds measured on a Sun Ultra Sparc 248 Mhz, using `gcc` 2.8.1 with the `-O2` option, both for `ms-wam` and for `wamcc`.

Program	ms-wam 1.0	wamcc 2.22	Sicstus 3.0 (emulated)	Sicstus 3.0 (native)
browse	0.190	0.550	0.760	0.240
fib	0.100	0.230	0.360	0.120
ham	0.240	0.590	0.550	0.230
nrev	0.820	3.370	2.480	0.520
qsort	0.300	0.610	0.630	0.240
queens(16)	0.150	0.320	0.530	0.140
queens_n(8)	0.040	0.100	0.100	0.040
queens_n(10)	0.660	1.980	1.730	0.600
query	0.140	0.390	0.560	0.340
tak	0.030	0.060	0.100	0.030
zebra	0.020	0.040	0.040	0.020
average speed up of ms-wam		2.6	3.0	1.1

Table 1. `ms-wam` versus `wamcc` and Sicstus

The most important and relevant comparison is between `wamcc` and `ms-wam`. Our system was constructed using the C implementation of the WAM from `wamcc`. Therefore the average speed-up of 2.6 indicates the performance gain of the multiple specialization strategy described in this paper.

Also important is the comparison with the de facto reference for performance of Prolog systems, the Sicstus compiler. On average, `ms-wam` is 10% faster than Sicstus 3.0 generating native code. Yet, the C generation of `ms-wam` has the advantage of running on multiple architectures as emulated Sicstus, over which it has an average speed-up of 3 times. Abandoning the C generation and directly generating native code will bring a considerable improvement on the performance of `ms-wam`, achieving the performance of the Aquarius system.

In table 2 we present results for the code growth of our multiple specialization technique. We present the size in Kbytes of the object file generated by `gcc` and the executable file after linking, both for `wamcc` and for `ms-wam`.

Program	ms-wam 1.0 object file	wamcc 2.22 object file	ms-wam 1.0 executable file	wamcc 2.22 executable file
browse	60.1	33.7	358.9	339.7
fib	6.5	7.8	324.7	305.8
ham	43.8	31.3	348.3	335.9
nrev	23.1	16.6	320.0	312.5
qsort	27.8	14.5	342.7	310.1
queens(16)	17.6	12.1	316.1	308.7
queens_n(8)	29.8	17.1	326.4	310.7
queens_n(10)	29.8	17.1	326.4	310.7
query	26.1	26.5	341.1	337.1
tak	12.7	8.0	312.0	305.7
zebra	43.1	19.6	347.0	328.9
Code growth	1.55		1.04	

Table 2. Code growth

The code expansion of our multiple specialization technique is quite modest, representing less than 5% in the executable files sizes and about a 1.5 expansion factor in the object files. This is due to the smaller implementation of the specialized predicates, which balances the larger number of predicates. In some cases, where the number of versions created is small, the multi-specialized object file is even smaller than the `wamcc` file (fib and query).

6 Conclusions

Program specialization will always depend on program analysis. Therefore, simplification of program specialization will always depend on analysis simplification. Local analysis techniques are much more simpler than global analysis techniques. The main effort of our work was to achieve the global analysis benefits through local analysis. Our multiple specialization strategy is the way to propagate local analysis information through procedure calls, allowing global program optimization.

The local analysis we perform is much easier to do at the WAM level. In fact, part of this analysis is already done in the Prolog to WAM compilation, giving rise to specialized instructions. We reuse this analysis promoting its instruction scope to an inter-procedural and global analysis.

The complexity of our `ms-wam` system is not higher than the complexity of the `wamcc` basic system, a very readable and extendable compiler. In spite of this simplicity, we out-perform native Sicsus 3.0. This is even more impressive because we are generating C code as target language. Thus we are not limited to the Sparc architecture as Sicsus, running in every architecture which has a C compiler.

The purpose of this work is not to be an alternative to the abstract interpretation techniques in order to achieve code specialization. Abstract interpretation

techniques are used in our system. We plan to extend the optimizations we perform by using more abstract interpretation algorithms. However, it is important to simplify such algorithms, simplifying, for instance, the analysis domains. Our analysis of exit types is simple because it is just an exit types analysis. What makes this kind of analysis algorithms complex and expensive is the association of exit and entry types analysis. The unfolded Prolog source code generated will also allow important simplification of abstract interpretation algorithms.

The integration of our multiple specialization technique into a Prolog compiler is simple. In fact, a relevant aspect of this specialization technique is that it is done at the implementation level. It depends on the process of compilation and in the generation of WAM code. The intermediate language that is generated is almost WAM code, complemented with some type and mode information. This allows the instructions of this intermediate language to be easily implemented in the WAM engine of the original compiler.

References

- [1] Hassan Ait-Kaci. *Warren's Abstract Machine — A Tutorial Reconstruction*. MIT Press, 1991.
- [2] M. Bruynooghe and G. Jenssens. An instance of abstract interpretation integrating type and mode inferencing. In *Proc. of ICSLP'88*, pages 669–683, Seattle, Washington, 1988. MIT Press.
- [3] Philippe Codognet and Daniel Diaz. wamcc: Compiling Prolog to C. In *Proc. of ICLP'95*. MIT Press, 1995.
- [4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of SPPL'77*, pages 238–252, Los Angeles, California, 1977.
- [5] S. K. Debray and D. S. Warren. Automatic mode inference for prolog programs. *The Journal of Logic Programming*, 5(3):78–88, September 1988.
- [6] D. Gudeman, K. De Bosschere, and S. Debray. jc: An efficient and portable sequential implementation of janus. In *Joint International Conference and Symposium on Logic Programming*, Washington, 1992. "MIT Press".
- [7] B. Haussman. Turbo Erlang: Approaching the Speed of C. In Evan Tick, editor, *Implementations of Logic Programming Systems*. Kluwer, 1994.
- [8] M. Hermenegildo, R. Warren, and S. K. Debray. Global Flow Analysis as a Practical Compilation Tool. *JLP*, 13(4):349–367, August 1992.
- [9] D. Jacobs and A. Langen. Accurate and efficient approximation of variable aliasing in logic programs. In *Proc. NACLP'89*, pages 154–165. "MIT Press", 1989.
- [10] A. Krall and U. Neumerkel. The Vienna Abstract Machine. In *Proc. of PLIP'90*, number 456 in LNCS, pages 121–135. Sweden, Springer-Verlag, 1990.
- [11] C. S. Mellish. Abstract interpretation of prolog programs. In *Proc of ICLP'86*, number 225 in Lecture Notes in Computer Science. Springer-Verlag, 1986.
- [12] S. Peyton Jones, T. Nordin, and D. Oliva. C-: a portable assembly language. In *[To appear in the proceedings of IFL'97.]*, St. Andrews, Scotland, 1997.
- [13] J. Tan and I. Lin. Compiling dataflow analysis of logic programs. In *Proc. of PLDI'92*, pages 106–115, San Francisco, California, 1992. SIGPLAN.
- [14] A. Taylor. *High Performance Prolog Implementation*. PhD thesis, University of Sydney, June 1991.

- [15] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, November 1990.
- [16] P. Van Roy. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *The Journal of Logic Programming*, 19/20, May/July 1994.
- [17] David H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.
- [18] W. H. Winsborough. Path-dependent reachability analysis for multiple specialization. In *Proc. of NACLP'89*. "MIT Press", 1989.

A Flexible Framework for Dynamic and Static Slicing of Logic Programs^{*}

Wamberto Weber Vasconcelos^{**}

Department of Statistics & Computing
State University of Ceará
Ceará, Brazil
wvasconcelos@acm.org

Abstract. Slicing is a technique for automatically obtaining subparts of a program with a collective meaning. When the slicing takes into account the actual execution of the program, it is said to be *dynamic*; when only statically available information is used, it is said to be *static*. In this paper we describe a flexible framework to slice logic programs that accommodates both dynamic and static approaches naturally. Our framework addresses practical aspects of logic programs such as different executions, system predicates and side-effects and can be customized to any logic language, provided we have its (concrete or abstract) interpreter and information on the behavior of system predicates.

Keywords. Logic program slicing, program analysis & understanding.

1 Introduction

Slicing is a technique for obtaining subparts of a program with a collective meaning. It was originally proposed by Weiser [35, 36], in the context of procedural languages. When we slice a program, we are interested in automatically obtaining those portions of the program responsible for specific parts of the computation. Program slicing is a useful technique for program development and reuse. Slicing can be particularly useful when debugging programs [35].

We have investigated the adequacy of some slicing techniques, originally proposed for procedural languages, for slicing logic programs. Our efforts are aimed at obtaining *executable* slices, *i.e.*, the slices must reproduce parts of the original computations. This is in contrast to methods whose slices produced are simply subsets of commands from the original program and do not necessarily comprise an executable program. Let us consider program *s.c/3* below,

^{*} Work partially carried out while the author was a Visiting Researcher at the Institut für Informatik, Universität Zürich, Switzerland.

^{**} Partially sponsored by the Brazilian National Research Council (CNPq) grant no. 352850/96-5.

```

s_c(L, Sum, Count) ←
    s_c(L, 0, Sum, 0, Count)

s_c([], S, S, C, C) ←
s_c([X|Xs], AcS, S, AcC, C) ←
    NAcS is AcS + X,
    NAcC is AcC + 1,
    s_c(Xs, NAcS, S, NAcC, C)

```

which sums (second argument) and counts (third argument) the elements of a list (first argument); *s_c/3* makes an initial call (with appropriate values) to *s_c/5* which employs two *accumulator pairs* [29]. Three of its slices are:

```

s_c(L, _, _) ←
    s_c(L, _, _, _, _)

s_c([], _, _, _, _) ←
s_c([X|Xs], _, _, _, _) ←
    s_c(Xs, _, _, _, _)

```

```

s_c(L, Sum, _) ←
    s_c(L, 0, Sum, _, _)

s_c([], S, S, _, _) ←
s_c([X|Xs], AcS, S, _, _) ←
    NAcS is AcS + X,
    s_c(Xs, NAcS, S, _, _)

```

```

s_c(L, _, Count) ←
    s_c(L, _, _, 0, Count)

s_c([], _, _, C, C) ←
s_c([X|Xs], _, _, AcC, C) ←
    NAcC is AcC + 1,
    s_c(Xs, _, _, NAcC, C)

```

The underscore symbols “*_*” stand for *anonymous variables* [7, 29], *i.e.* place holders for arguments which are not important for the computation. Those argument positions which are not relevant to the slice are replaced by anonymous variables, an approach introduced in [26]. All the slices above are fully operational programs that compute parts of the original program: if we pose them queries, a subset of the results computed by the original program is obtained.

We have adapted some of the ideas for slicing programs in procedural languages and proposed a flexible framework for logic programs. We regard our approach as “flexible” because it can be customized to any logic language, provided we have its (concrete or abstract) interpreter and information on the behavior of system (built-in) facilities.

In the next Section we survey some of the work on program slicing, mostly concerned with programs written in procedural languages, but we also study existing research on logic program slicing. In Section 3 we outline our proposed framework and propose a criterion for logic program slicing. In Section 4 we introduce a means to represent the meaning of logic programs, necessary when we want to slice them. Section 5 describes our proposal for detecting and formalizing different kinds of dependences within logic programs so as to allow the removal of those irrelevant parts of a program. In Section 6 we formalize a practical notion of *relevance* between constructs of a logic program. Section 7 describes a slicing algorithm for our framework. In Section 8 we draw conclusions and make final remarks on the work described here.

In this paper, variables are denoted as x, y , predicate symbols as p^n, q^n and functors as f^n, g^n (where n is the arity), and terms as s, t . To allow for a homogeneous treatment of head and body goals, we will assume that our programs are *normalized*, that is, their clauses only contain head goals of the form $p^n(x_0, \dots, x_n)$ and their body goals are of the form $p^n(x_1, \dots, x_n)$, $x = y$ or $x = f^n(y_1, \dots, y_n)$, where x, x_i, y, y_j are all distinct variables.

2 Program Slicing

Weiser originally proposed in [35] the notion of program slicing for procedural languages, giving psychological evidence that programmers, consciously or not, break down large programs into smaller coherent pieces. Functions, subroutines, and so on, are all *contiguous* portions of the original program with particular purposes which are easily observed. However, another form of program pieces can be detected and used by experienced programmers when analyzing a program: these are sets of statements related by the flow of data, not necessarily contiguous but scattered through the program.

Program slices can be seen more generally as those parts of a program that potentially affect the values computed at some point of interest, referred to as the slicing criterion [30]. Following the work by Weiser, extensive research has been done (see [4, 30] for an overview of the field). Research has been carried out to improve the quality (the smaller, the better) of the obtained slices (*e.g.* [12, 18]) with different methods and techniques; to minimize the computational costs of obtaining slices (*e.g.* [24]); to generalize slicing algorithms (*e.g.* [17, 18, 28]); to address special features of programming languages (*e.g.* [1, 16]); and to investigate applications of slicing (*e.g.* [10, 12]).

Slicing methods all have a same general framework: information on the data- and control flow of the program to be sliced is used to verify the *relevance* of each statement of the program to the chosen criterion. The definition of relevance and the way it is computed depend on the available information and both offer dimensions to study and compare slicing methods. Depending on the application of the slices and the programming language issues being tackled (procedures, arbitrary data types, etc.) different solutions may be necessary.

When the slicing technique, in order to gather information on the data- and control flow of the program, takes into account an actual and specific execution (or set of executions) of it then it is said to be *dynamic*. When the slicing uses only statically available information then it is said to be *static* [30].

2.1 Criteria for Program Slicing

Slicing of programs is performed with respect to some criterion. Weiser [35, 36] proposes as a criterion the number i of a command line and a subset V of program variables. According to this criterion, a program is analyzed and its commands are checked for their relevance to command line i and those variables in V ; a command is relevant if it tests or assigns values to variables in V or to variables not in V that might be used, directly or not, to test or assign values to variables in V . However, different criteria have been defined by other authors: for instance, in [20] we find a criterion geared towards dynamic slicing.

2.2 Logic Program Slicing

Slicing of logic programs has only recently been addressed. In [31, 32] we are presented with a very first attempt at a particular form of slicing for logic programs. In that work, the slices obtained are possibly non-contiguous portions of an original Prolog program responsible for some of the computations, and they

may not necessarily be executable. The slices detected are called *programming techniques*: they correspond to the pieces of code used by programmers in a systematic way when developing their programs [5].

In [15] we are presented with a static slicing method, to be offered as part of a debugging tool. The method, however, is not aimed at producing slices (*i.e.*, subsets of commands) of the original program. Rather, it is used to slice the *proof-tree* of the logic program, eliminating those parts which are not related to the point of the program where a bug has been detected.

In [26] we are presented with a slicing algorithm to produce executable slices of Prolog programs. A useful contribution of that work is the introduction of *argument abstraction*: when analyzing the relevance of a subgoal the algorithm may decide that certain arguments are not relevant and should be left out. Irrelevant arguments are replaced with anonymous variables, thus *partially* keeping a program statement.

In [14] a generic approach is proposed to perform the slicing of programs written in any procedural, logic or functional programming language. By specifying the semantics of the programming language using *natural semantics*, a special formalism, it is possible to obtain slicing analyses of programs automatically.

2.3 Why Slicing of Logic Programs Is Difficult

In logic languages some constructs may have many different possible executions. Determining the actual or possible computations of a program is necessary when slicing it. Slicing of logic programs is more sophisticated since it must account for this diversity of meanings for the same program. Furthermore, in purely declarative logic languages there are no explicit references to the flow of execution and iteration can only be achieved via recursion. Hence, interprocedural calls play an essential role in logic programming. Slicing techniques designed to cope with interprocedural calls in procedural programming languages are more complex than those for self-contained programs [16, 30].

3 A Framework for Slicing Logic Programs

We have developed a framework for slicing logic programs. It incorporates ideas of slicing techniques geared towards procedural languages. We start with a logic program Π and a criterion \mathcal{C} (defined in Subsection 3.1), and initially obtain the meaning of Π (described in Section 4); by “meaning” here we refer to a procedural interpretation of the program. Using our representation for the meaning of the program we build a term dependence graph (Section 5) which captures relationships among terms of a program. A notion of *relevance* of program constructs with respect to criteria is formally defined (Section 6): it uses *both* the represented meaning of the program and the graph. A slicing algorithm is finally presented (Section 7): a slice Π' is a variant of Π in which some subgoals may have been removed, argument positions may have been replaced for anonymous variables, and all those predicates that are not relevant to the criterion have been deleted.

3.1 Criteria for Logic Program Slicing

We have extended the usual syntax of logic programs in order to enable the unambiguous reference of any literal of a program. We have adopted a simple identification system by means of which each subgoal of a program is assigned a label $\langle i, j \rangle$, $i \geq 1, j \geq 0$, where i is the order in which the clause appears in the source program, and j is the order of the subgoal within its clause: the head goal has order 0 and the body goals have order 1, 2, and so on. Without loss of generality, we will assume that our logic programs have this unique identification for each subgoal. We will also refer to the uniquely defined subgoals as *literal positions*. To provide for a homogeneous treatment, we will assume that a query G_1, \dots, G_n has positions $\langle 0, i \rangle$, $1 \leq i \leq n$, respectively, for its literals.

In order to refer in a unique way to any term of a literal within a logic program, we will employ the concept of *argument position* (or simply *position*) adapted from [6]:

Definition 1. Let $\langle c, i \rangle$ be the identification of the i -th literal $p_j^r(t_1, \dots, t_r)$ of the c -th clause in Π , then its k -th *argument position* (or simply *position*) is uniquely defined within Π by the tuple $\langle c, i, p_j^r, k \rangle$.

We will refer generically to argument positions as α and β . The slicing criterion proposed in [32] consists of a query and an argument position in that query. Only the top most predicate (that is, the query predicate itself) is actually sliced with respect to the given argument position. In [15] and [26] the criterion consists of an argument position of one of the subgoals in the program. Additionally, in [26] a *success leaf* in the SLD-derivation of the query is also supplied as part of their criterion. We have defined the following notion of criterion:

Definition 2. A *criterion* \mathcal{C} is the pair $\langle (G_0, \dots, G_n), P \rangle$ where G_0, \dots, G_n is a sequence of subgoals and P is a set of argument positions of Π .

The positions of our criterion can be partially specified, with universally quantified variables used instead of actual values. For instance, $\langle x, 0, p_j^r, y \rangle$ denotes all positions of those head goals of all clauses defining predicate p_j^r . We have deliberately left out from our definition of criterion the *success leaf* concept introduced in [26], for two reasons: *a)* they are not intuitive components of a program execution, that is, it is not realistic to assume or believe that programmers are aware of the proof-tree of a program and its contents; and *b)* if slicing is performed because the program fails and debugging is needed, then it may not make sense to ask for a success leaf of the proof-tree for that program.

4 Mode-Annotated Logic Programs

We have proposed a means to formally represent the *procedural* meaning of a logic program, *i.e.*, the way the program is *executed*. Proof-trees [3, 21, 29] and traces [7, 29] for Prolog programs, for instance, concentrate on dynamic aspects of program execution, showing the progress of the computation as unifications taking place. However, in those representations once a unification takes place we lose information on what terms *used* to be. We propose a means to incorporate

the procedural meaning of logic programs into their syntax. This proposal consists of representing the instantiation modes of variables before and after each subgoal, in explicit *mode annotations*. Our thesis here is that the computations of logic program constructs can be conveniently represented by the instantiations of variables taking place. This should not be seen as controversial for the actual computations of a logic program are, in fact, performed in terms of changes in the instantiation of their variables [2, 27].

In our approach, we relate to each variable a token describing its instantiation status or mode: “f” for *free* variables; “g” for *ground* variables, *i.e.* variables bound to constants or composed terms with ground subterms only; and “i” for *instantiated* variables, *i.e.* variables *not free* (*i.e.*, instantiated to partially ground terms). These tokens are an attempt to move from particular values of variables to a more abstract setting in which *sets of values* can be considered instead¹. The computations of logic program constructs are represented by the changes taking place in their variables, recorded in mode-annotations:

Definition 3. A *mode-annotation* θ is a finite (possibly empty) set of pairs x/m where x is a variable and $m \in \{\mathbf{f}, \mathbf{g}, \mathbf{i}\}$ is the *instantiation mode* of x .

Each subgoal of a program execution has two associated mode-annotations, one recording the instantiation of its variables *before* its execution, and another *after*:

Definition 4. A *mode-annotated goal* \tilde{G} is the triple $\theta G \theta'$ where G is a subgoal and θ, θ' are mode-annotations with elements x/m , x being a variable in G .

Our goal mode-annotations should enable us to detect *changes* that have happened to the instantiation of variables. The detection of changes allows us to represent the term dependences of a program. We employ the definition of annotated subgoals to define mode-annotated executions:

Definition 5. A *mode-annotated execution* \tilde{E} is a finite (possibly empty) sequence $\tilde{G}_0, \dots, \tilde{G}_n$ of mode-annotated subgoals $\tilde{G}_i, 0 \leq i \leq n$.

Both concrete and abstract interpreters can be used to obtain mode-annotated executions, with advantages and drawbacks, as explained in Subsection 4.1 below. When a concrete interpreter is employed, then the slicing is *dynamic*; if abstract interpretation [8] is used then the slicing is said *static*. We show below the concrete mode-annotated execution for query `List = [1,23,5], s_c(List,SL,CL)` using a normalized version of the *s_c/3* program of Section 1 (with literal positions):

¹ Different sets of tokens have been proposed for specific applications. For instance, [22] uses the set $\{\mathbf{g}, \mathbf{ng}\}$ (ground and non-ground, respectively) to analyze the flow of Prolog programs and [9] use the set $\{\mathbf{e}, \mathbf{c}, \mathbf{f}, \mathbf{d}\}$ (respectively: empty, closed or ground, free and don't know) for mode inferencing.

$\{List/g, SL/f, CL/f\}$	$\langle 0, 1, List = [1, 23, 5] \rangle$	$\{List/g\}$
$\{L/f, Sum/f, Count/f\}$	$\langle 0, 2, s_c(List, SL, CL) \rangle$	$\{List/g, SL/g, CL/g\}$
$\{A/f\}$	$\langle 1, 0, s_c(L, Sum, Count) \rangle$	$\{L/g, Sum/f, Count/f\}$
$\{B/f\}$	$\langle 1, 1, A = 0 \rangle$	$\{A/g\}$
$\{L/g, A/g, Sum/f, B/g, Count/f\}$	$\langle 1, 2, B = 0 \rangle$	$\{B/g\}$
$\{G/f, AcS/f, S/f, AcC/f, C/f\}$	$\langle 1, 3, s_c(L, A, Sum, B, Count) \rangle$	$\{L/g, A/g, Sum/g, B/g, Count/g\}$
$\{G/g, X/f, Xs/f\}$	$\langle 3, 0, s_c(G, AcS, S, AcC, C) \rangle$	$\{G/g, AcS/g, S/f, AcC/g, C/f\}$
$\{NacS/f, AcS/g, X/g\}$	$\langle 3, 1, G = [X Xs] \rangle$	$\{G/g, X/g, Xs/g\}$
$\{H/f\}$	$\langle 3, 2, NacS \text{ is } AcS + X \rangle$	$\{NacS/g, AcS/g, X/g\}$
$\{NacC/f, AcC/g, H/g\}$	$\langle 3, 3, H = 1 \rangle$	$\{H/g\}$
$\{Xs/g, NacS/g, S/f, NacC/g, C/f\}$	$\langle 3, 4, NacC \text{ is } AcC + H \rangle$	$\{NacC/g, AcC/g, H/g\}$
$\{G_1/f, AcS_1/f, S_1/f, AcC_1/f, C_1/f\}$	$\langle 3, 5, s_c(Xs, NacS, S, NacC, C) \rangle$	$\{Xs/g, NacS/g, S/g, NacC/g, C/g\}$
$\{Xs/g, NacS/g, S/f, NacC/g, C/f\}$	$\langle 3, 0, s_c(G_1, AcS_1, S_1, AcC_1, C_1) \rangle$	$\{G_1/g, AcS_1/g, S_1/f, AcC_1/g, C_1/f\}$
$\{D/f, S_2/f, E/f, C_2/f, F/f\}$	$\langle 3, 5, s_c(Xs, NacS, S, NacC, C) \rangle$	$\{Xs/g, NacS/g, S/g, NacC/g, C/g\}$
$\{D/g\}$	$\langle 2, 0, s_c(D, S_2, E, C_2, F) \rangle$	$\{D/g, S_2/g, E/f, C_2/g, F/f\}$
$\{S_2/g, E/f\}$	$\langle 2, 1, D = [] \rangle$	$\{D/g\}$
$\{C_2/g, F/f\}$	$\langle 2, 2, S_2 = E \rangle$	$\{S_2/g, E/g\}$
	$\langle 2, 2, C_2 = F \rangle$	$\{C_2/g, F/g\}$

Subgoals are shown in the order they appeared during the execution, however, this ordering is not relevant for the slicing. Subscripts are used to differentiate distinct variables with the same name. All subgoals with goal number 0 are head goals, and whenever they appear we can infer that a match between a subgoal and the head goal of a clause has taken place. When $\theta_G G \theta'_G$ matches head goal $\theta_H H \theta'_H$, θ_G and θ_H have the modes of variables *before* the matching, θ'_H has the modes of variables of H *after* the matching and θ'_G the modes of the variables of G after its complete execution. Only one record is kept for each goal/head goal match. In the example above, subgoal $\langle 3, 5 \rangle$ matches head goal $\langle 3, 0 \rangle$ twice, but since the modes of variables are respectively the same, only one record is kept.

A natural alternative to mode-annotations could be the actual unifications of the execution, that is, sets of pairs x/t where x is a variable and t is a term. However, in a program execution, a subgoal can be executed many times with different terms associated to its variables. These different instances offer no extra information: the relationships among argument positions, as we shall see in Section 5 below, will simply be re-stated with different terms. If the actual values of terms in substitutions are replaced by their modes, this enhances the similarities between annotated goals and equivalent constructs can be omitted. An extra benefit of mode-annotations is that they can also be obtained by abstract interpretation, thus giving extra flexibility to our framework.

4.1 Concrete and Abstract Mode-Annotated Programs

Mode-annotated executions of a program can be obtained either by concrete or abstract interpretation. In both cases, it is a straightforward procedure to augment the respective meta-interpreters so as to enable them to get mode-annotated executions, as demonstrated in [32]. However, both approaches have advantages and drawbacks. A concrete interpreter, on one hand, provides accurate mode-annotations. However, since the concrete interpreter actually runs the program, if the program loops, then the meta-interpreter also loops. Furthermore, if certain clauses of the program are not used in the execution, then these clauses would not appear in the mode-annotated execution. An abstract interpreter, on the other hand, provides *safe* approximations of the actual computations taking place [8, 19, 23]. To cope with inherent imprecisions of abstract

interpretation, an extra token “?” is needed, which stands for an *unknown* instantiation status. The drawbacks of the concrete interpreter are avoided: since the program is not actually run, then there is no risk of loops; all clauses of the program are analyzed, provided they are reachable by some computation [8]. However, a disadvantage of the abstract interpreter with respect to the concrete interpreter is the low quality of its mode-annotations: tokens “i” and “?” frequently appear in mode-annotations obtained via abstract interpretation which, if obtained via a concrete interpreter, would be represented more accurately. Extra goal/head goal unifications may also appear during the abstract interpretation of a program, which would not happen otherwise.

5 Term Dependence Graph

Our representation of the procedural meaning of logic programs above enables us to detect and formalize the relationships between terms of a program. We propose that a graph be used to represent two important kinds of information, *viz.* the *data flow* among terms, that is, how data are passed on from one term to another, and the *control flow* of a program, that is, those points of a program where a computation affecting the flow of execution is performed:

Definition 6. A *term dependence graph* \mathcal{G} is a finite (possibly empty) set of edges $x \leftarrow t$ and nodes $test(t)$, where x and t are, respectively, a variable and a term of a mode-annotated program execution \tilde{E} .

Edges depict the flow of data and *test* nodes represent those terms tested during the program execution. Our graph is built by examining the program execution: each mode-annotated subgoal gives rise to a possibly empty graph representing the information extracted from its computations. Given a mode-annotated execution, its graph is the union of all sub-graphs of its subgoals.

Unifications and system predicates whose possible meanings are known in advance are employed to build the graph. We thus need to know, in advance, the set of system predicates offered by the language and their possible meanings. We have defined a framework to represent the meaning of system predicates in terms of edges and nodes in the graph. This framework can be customized to any language. We have employed Prolog’s set of system predicates [7, 29].

Slicing techniques must be *safe*: in the absence of accurate information all possibilities must be considered. When mode-annotations get less precise, with more “i” and “?” tokens, all possible computations must be accounted for. Given a mode-annotated subgoal, our approach tries all possible formulae to build graphs: some edges or nodes may overlap, but the associated graph is always extended. By adding new edges and nodes to the associated graph, all possible different computations are addressed and the relationships they forge among terms of a program are formally represented.

5.1 Unique Term Identification: Term Labeling

For our purposes of slicing a logic program, it is necessary that each term of a mode-annotated execution be uniquely identified. This is required to correctly

map a term onto its argument position. It happens automatically with variables: fresh variables are created for each new clause employed. However, it is not always possible to precisely tell the position of an atom – in the mode-annotated execution for *s_c/3* shown above, for instance, it is not possible to tell the position of “0” for it is found in two different places. To solve this, we propose that each term of a program which is not a variable be assigned a unique label $\mathbf{t}_0, \mathbf{t}_1, \dots$

5.2 Detecting Potential Changes via Modes

In order to build the term dependence graph we need to know which computations may have taken place in each subgoal. This can be inferred via the mode-annotations of each variable in the subgoal and how they change before and after the subgoal.

The tokens associated to a variable of a mode-annotated subgoal provide information about possible changes taking place. Since our framework also handles mode-annotated executions obtained via abstract interpretation (and these may contain imprecise, albeit safe, “i” and “?” tokens) it is only possible to formalize *potential* changes: for instance, if a variable has modes “?” associated with it both before and after a subgoal, it may not be possible to say whether or not changes actually took place. We formalize a definition for potential changes below:

Definition 7. Relation $changes(x, \theta, \theta')$, where $x/m \in \theta, x/m' \in \theta'$, holds iff *a*) ($m = \mathbf{f}$ and $m' \neq \mathbf{f}$); or *b*) $m = \mathbf{i}$; or *c*) $m = ?$.

For accurate mode-annotations the definition above will correctly infer actual changes; for less accurate mode-annotations, potential changes that may have happened will be inferred.

5.3 Data Flow between Terms

The meaning of system predicates can be detected by examining the unifications that took place and are formalized as edges/nodes of the graph. An edge $x \leftarrow t$ indicates that a computation made data flow from t onto x . We have formalized a relationship $graph_G$ mapping a mode-annotated goal to its set of edges/nodes, given some restrictions. A possible case of unification, for instance, is depicted by the formula

$$graph_G(\theta\langle c, l, x = y \rangle \theta', \{x \leftarrow y\}) \leftarrow changes(x, \theta, \theta') \wedge \neg changes(y, \theta, \theta')$$

That is, in a subgoal $x = y$, if x may have changed and y did not change, then we can infer that data flowed from y to x . Similar definitions would be required for all system predicates: since the same predicate can be used in different contexts, all possible behaviors should be explicitly dealt with at this point. In [33] we provide a comprehensive list for Prolog system predicates and their associated graphs.

5.4 Control Flow Information

The control flow of a program is established by those computations that change or define the flow of execution. A node $test(t)$ indicates that a computation

performed a test making use of t , hence affecting or defining the execution. We show below the case when the $=/2$ predicate tests a variable x with a labeled term \mathbf{t} :

$$graph_G(\theta\langle c, l, x = \mathbf{t} \rangle\theta', \{test(x), test(\mathbf{t})\}) \leftarrow x/\mathbf{f} \notin \theta$$

The unification condition must hold to ensure that a test is being performed. We show, in Figure 1, the graph associated to the mode-annotated *s_c* program given above. Another important computation we are able to formalize concerns

θ_0	$\langle 0, 1, List = \mathbf{t}_0 \rangle$	θ'_0	$List \leftarrow \mathbf{t}_0,$
θ_1	$\langle 0, 2, s_c(List, SL, CL) \rangle$	θ'_1	$L \leftarrow List, SL \leftarrow Sum, Sum \leftarrow SL,$
θ_2	$\langle 1, 0, s_c(L, Sum, Count) \rangle$	θ'_2	$CL \leftarrow Count, Count \leftarrow CL,$
θ_3	$\langle 1, 1, A = \mathbf{t}_1 \rangle$	θ'_3	$A \leftarrow \mathbf{t}_1,$
θ_4	$\langle 1, 2, B = \mathbf{t}_2 \rangle$	θ'_4	$B \leftarrow \mathbf{t}_2,$
θ_5	$\langle 1, 3, s_c(L, A, Sum, B, Count) \rangle$	θ'_5	$G \leftarrow L, AcS \leftarrow A, S \leftarrow Sum, Sum \leftarrow S,$
θ_6	$\langle 3, 0, s_c(G, AcS, S, AcC, C) \rangle$	θ'_6	$AcC \leftarrow B, C \leftarrow Count, Count \leftarrow C,$
θ_7	$\langle 3, 1, G = [X Xs] \rangle$	θ'_7	$test(G), X \leftarrow G, Xs \leftarrow G,$
θ_8	$\langle 3, 2, NAcS \text{ is } AcS + X \rangle$	θ'_8	$NAcS \leftarrow AcS, NAcS \leftarrow X,$
θ_9	$\langle 3, 3, H = \mathbf{t}_3 \rangle$	θ'_9	$H \leftarrow \mathbf{t}_3,$
θ_{10}	$\langle 3, 4, NAcC \text{ is } AcC + H \rangle$	θ'_{10}	$NAcC \leftarrow AcC, NAcC \leftarrow H,$
θ_{11}	$\langle 3, 5, s_c(Xs, NAcS, S, NAcC, C) \rangle$	θ'_{11}	$G_1 \leftarrow Xs, AcS_1 \leftarrow NAcS, S_1 \leftarrow S, S \leftarrow S_1,$
θ_{12}	$\langle 3, 0, s_c(G_1, AcS_1, S_1, AcC_1, C_1) \rangle$	θ'_{12}	$AcC_1 \leftarrow NAcC, C_1 \leftarrow C, C \leftarrow C_1,$
θ_{13}	$\langle 3, 5, s_c(Xs, NAcS, S, NAcC, C) \rangle$	θ'_{13}	$D \leftarrow Xs, S_2 \leftarrow NAcS, E \leftarrow S, S \leftarrow E,$
θ_{14}	$\langle 2, 0, s_c(D, S_2, E, C_2, F) \rangle$	θ'_{14}	$C_2 \leftarrow NAcC, F \leftarrow C, C \leftarrow F,$
θ_{15}	$\langle 2, 1, D = \mathbf{t}_4 \rangle$	θ'_{15}	$test(D), test(\mathbf{t}_4),$
θ_{16}	$\langle 2, 2, S_2 = E \rangle$	θ'_{16}	$E \leftarrow S_2,$
θ_{17}	$\langle 2, 2, C_2 = F \rangle$	θ'_{17}	$F \leftarrow C_2$

Fig. 1: Mode-Annotated Program and Associated Term Dependence Graph

the matching of a subgoal with head goals. In the formula below a more complex construct is employed to describe its context, since two literals come into play in the computation, the subgoal and the head goal:

$$graph_G\left(\frac{\theta(l, m, p(x_1, \dots, x_n))\theta'}{\theta\langle n, 0, p(y_1, \dots, y_n) \rangle\theta''}, G\right) \leftarrow \bigwedge_{i=1}^n graph_G(\theta\langle \leftarrow, \neg, x_i = y_i \rangle\theta'', G_i) \wedge G = \bigcup_{i=1}^n G_i$$

This is the only *graph* definition in which two annotated subgoals are employed: in order to improve its visualization we have enclosed each annotated subgoal in a box. We are able to precisely characterize the matching of a subgoal with the head goal of a clause — the latter ought to have a subgoal ordering 0, as in $\langle n, 0 \rangle$. The computations arising from this matching are the union of all those unifications of corresponding subterms in the literals. The computations taking place during the goal/head-goal matching, depicted above, define the distinct relationships among variables of different clauses. This is referred to in [15] and in [26] as *external dependences*.

5.5 Side Effects, Data-, and Control Flow

Side-effects are important components of logic programs. System input/output predicates, for instance, are executed by means of such side-effects. The actual input/output taking place in the execution of such predicates is of a strictly procedural nature, making it difficult to provide a clean and accurate representation for such phenomena. We have devised a way to represent side-effects, germane to our needs.

The notion of data flow we have captured involves two terms. However, some of the input/output predicates we want to address may only have one argument position. To circumvent this difficulty, we propose here to employ two special-purpose terms. These terms, denoted by *input* and *output*, are a reference to the fact that a side-effect has been used in the proof of the subgoal. For instance, the *write/1* predicate has associated formula $graph_G(\theta\langle c, l, \mathbf{write}(x) \rangle \theta', \{output \leftarrow x\})$ establishing its computation – the associated edge indicates that data flows from x onto the *output* term. In order to allow the slicing to be performed with respect to the side-effects of a program, we have defined two extra “dummy” argument positions, $\langle 0, 0, input, 0 \rangle$ and $\langle 0, 0, output, 0 \rangle$. The slicing of logic programs can be geared towards always considering the *input* and *output* argument positions as part of the criterion or it may only consider them as part of the criterion when explicitly told. We have adopted the former approach.

6 Relevance of Argument Positions

Our approach to logic program slicing consists of analyzing the argument positions of a program and testing their *relevance* to one of the argument positions of the criterion. Our definition of relevance is stated in terms of argument positions but uses the term dependence graph, however, it is possible to obtain all those terms occupying a certain argument position, by scanning each subgoal:

Definition 8. Relation $terms(\alpha, \tilde{\mathcal{E}}, \{t_0, \dots, t_n\})$ holds iff terms $\{t_0, \dots, t_n\}$ occupy argument position α in $\tilde{\mathcal{E}}$

It might be argued that the graph could have been proposed using argument positions as nodes, rather than the terms of a program. However, it is essential, in order to detect transitivity relations in the graph, to employ the terms themselves. Furthermore, it is simpler and more natural to describe the phenomena of program executions using the terms of the program.

A notion of relevance similar to the one described here was originally proposed in [32] and we have adapted it to deal with argument positions in program executions:

Definition 9. Given $\tilde{\mathcal{E}}$ and \mathcal{G} , α is relevant to β if one of the cases below holds:

$$\begin{aligned} relevant(\alpha, \beta, \tilde{\mathcal{E}}, \mathcal{G}) &\leftarrow relevant(\alpha, \gamma, \tilde{\mathcal{E}}, \mathcal{G}) \wedge relevant(\gamma, \beta, \tilde{\mathcal{E}}, \mathcal{G}) \\ relevant(\alpha, \beta, \tilde{\mathcal{E}}, \mathcal{G}) &\leftarrow terms(\alpha, \tilde{\mathcal{E}}, \{\dots t_\alpha \dots\}) \wedge terms(\beta, \tilde{\mathcal{E}}, \{\dots t_\beta \dots\}) \wedge t_\beta \leftarrow t_\alpha \in \mathcal{G} \\ relevant(\alpha, \beta, \tilde{\mathcal{E}}, \mathcal{G}) &\leftarrow terms(\alpha, \tilde{\mathcal{E}}, \{\dots t_\alpha \dots\}) \wedge test(t_\alpha) \in \mathcal{G} \end{aligned}$$

The first formula states that α is relevant to β if there is an intermediate argument position γ which α is relevant to, and also is relevant to β . This case formalizes the transitivity of the *relevant* relationship and amounts to finding out whether there are sequences of relevant intermediate terms.

The second formula exploits the immediate relevance between two argument positions: α is relevant to β if there are terms t_α and t_β occupying, respectively, positions α and β , such that an edge in \mathcal{G} states that data flowed from t_α to t_β .

The third formula holds if there is a node $test(t_\alpha)$ in \mathcal{G} such that t_α is in α . This means that α satisfies the relationship if a term in α is used in a computation that may define or alter the flow of execution. In such circumstances, the β position with respect to which α is being tested is not taken into account. Through this relationship all argument positions whose terms have been employed in computations that define or may alter the flow of control are relevant.

If all term dependences have been detected and appropriately represented, then the *relevance* relation above will correctly hold when α is relevant to β , since the terms associated with these positions will have their data- and control flow edges/nodes in the graph. The cost for this test is that of finding a path between two sets of nodes and the worst case arises when there are no paths. In such a circumstance, the cost is a polynomial function of the number of edges in the graph.

7 An Algorithm for Logic Program Slicing

We show in Figure 2 an algorithm for automatically obtaining a slice of a program Π with respect to a criterion \mathcal{C} . The slice output by the algorithm is another program Π' obtained by examining Π and checking the relevance of each of its positions α with respect to each position β of \mathcal{C} .

```

input program  $\Pi = \{C_1, \dots, C_m\}$  and criterion  $\mathcal{C} = \langle (G_0, \dots, G_n), P \rangle$ 
output program  $\Pi' = \{C'_1, \dots, C'_m\}$ 
begin
  1. obtain mode-annotated execution  $\tilde{\mathcal{E}}$  for  $\Pi \vdash (G_0, \dots, G_n)$ 
  2. obtain term dependence graph  $\mathcal{G}$  for  $\tilde{\mathcal{E}}$ 
  3. for each subgoal  $\tilde{G}$  in  $\tilde{\mathcal{E}}$  do for each  $\alpha$  in  $\tilde{G}$  do for each  $\beta$  in  $P$  do
      if relevant( $\alpha, \beta, \tilde{\mathcal{E}}, \mathcal{G}$ ) then  $\alpha$  is marked relevant
  4. for each subgoal  $G = \langle c, l, p(t_0, \dots, t_n) \rangle$  in  $\Pi$  do
      begin
        for each  $t_i$  do
          if there is a relevant  $\alpha$ ,  $terms(\alpha, \tilde{\mathcal{E}}, \{\dots t_i \dots\})$ , then  $t'_i = t_i$  else  $t'_i = \_$ 
          if there is  $t'_i \neq \_$  then  $G' = \langle c, l, p(t'_0, \dots, t'_n) \rangle$  else  $G' = true$ 
        end
      5. for each clause  $C_i = H \leftarrow G_1, \dots, G_n$  in  $\Pi$  do
          if  $H' = true$  then  $C'_i = true$  else  $C'_i = H' \leftarrow G'_1, \dots, G'_n$ 
      end
end

```

Fig. 2: Slicing Algorithm for Logic Programs

Steps 1 and 2 collect information on the program (its execution and term dependence graph). Step 3 examines the annotated subgoals of the program execution and performs a relevance analysis of each of its argument positions α with respect to $\beta \in P$: if α is found to be relevant to at least one β , then it is marked **relevant**; subsequent relevance tests will not change its status (an argument position is preserved if it is relevant to *at least* one argument position of the criterion). Step 4 checks each subgoal in Π and obtains a new sliced version

G' : terms t_i whose argument positions are **relevant** are preserved; those terms whose argument positions were not marked relevant are replaced with an anonymous variable “_”; if all terms of a subgoal have been replaced with anonymous variables then the corresponding sliced subgoal is a *true* predicate which always succeeds. Step 5 assembles the clauses of Π' using the sliced subgoals: if the head goal H has been sliced as *true* then the body will be empty; if H is not *true* then the body is assembled using the corresponding sliced subgoals.

The algorithm above can provide minimal slices, if a concrete interpreter is employed. If we have precise mode-annotations and a comprehensive set of formulae defining the meaning of all system predicates, then we can obtain all the actual relationships between the terms of a program and it is easy to see that the *relevant* test will correctly tell us which argument positions are relevant to argument positions of the criterion. However, the minimal slice obtained is particular to a set of specific executions. If, on the other hand, an abstract interpreter is used, then the slice is generic, but it may not be minimal for the mode-annotations lose precision and extraneous *test* nodes and data flow edges can be included in the term dependence graph. This is not surprising since it is proved in [36] that there are no algorithms to find the minimal slice of arbitrary programs. The computational costs of the algorithm above depend on the test for relevance between argument positions. This test is performed between the n argument positions of the program and the m argument position of the criterion, that is, $n \times m$ times.

It is not possible to compare the costs and accuracy of our slicing method with those of [26] because that work is not self-sufficient. Their proposal also relies on the accuracy of mode-annotations for the variables of a program but it is not mentioned how they are obtained nor how imprecisions are coped with. That slicing method relies further on a set of *failure positions*, that is, positions that can be responsible for a failure during the resolution of a goal. Their definition of failure positions, however, employs a notion of *search tree*, that is, the tree of an SLD-resolution proof of a query in a special format, but it is not mentioned how these are obtained nor whether the trees are explicitly handled. Their algorithm relies on the transitive closure of data dependences, an idea akin to that of finding a path between nodes of a graph, adopted in our approach.

Our framework, on the other hand, addresses how all necessary information should be obtained. We have implemented prototypical versions of the algorithm above in Prolog² and tested them on programs of medium complexity (*e.g.* the prototypes themselves, expert systems, assorted meta-interpreters and theorem provers and list processing programs). We observe that the performance of our prototypes depends on different factors, *viz.*, *a*) the size of the mode-annotated program and the number of its terms; *b*) the number of positions of the criterion; and *c*) the size and nature of the term dependence graph. As these factors grow, the performance of our prototypes slows down. Furthermore, if the graph has too many long chains of indirect relevance the slicing can also be slowed down. It should also be pointed out that, when a concrete interpreter is used to collect

² We have shown, for the sake of brevity, our slicing method as an algorithm.

the mode-annotations, the slicing obviously takes more time than the execution of the program to be sliced.

The accuracy of our prototypes also depends on different factors, *viz.*, the precision of the mode-annotations and the thoroughness and precision of the $graph_G$ definitions. We guarantee the thoroughness in our prototypes by employing a *catch-all* definition which create appropriate *test* nodes and edges using all terms of a system predicate without an associated $graph_G$ definition. This ensures that, in lack of precise information of a system predicate, safe assumptions are made.

In order to compare our results with those of [13, 14], we would have to formalize the semantics of Prolog, the logic programming language we adopted, using natural semantics. This is a complex task even for very simple languages, although to be performed only once. It remains unclear, however, how one can make sure that the semantics of a programming language formalized in the natural semantics is correct and this is essential for their approach. Contrastingly, the required customization for our framework relies on more practical aspects of the programming language, *viz.*, the syntax of system predicates and their possible usage modes.

8 Conclusions and Directions of Research

We have studied and compared existing techniques for slicing programs of procedural and logic languages and proposed a framework to slice logic programs. We regard our proposal as a flexible *framework*, rather than a slicing method specific to a particular logic language. The mode-annotated execution, concrete or abstract, can be obtained for any logic language, provided we have its interpreter; the term dependency graph is built straightforwardly if the meaning of the system predicates (built-ins) is available; finally, alternative definitions for *relevance* can be supplied. The slicing algorithm can be seen as being parameterized by all these components. A number of original techniques were devised and used:

- *Representation for procedural meaning of logic programs*: originally pursued in [32], we have extended it to cope with complete programs.
- *Definition of an expressive criterion*: we have defined an expressive notion of criterion for slicing logic programs.
- *A means to detect and represent data- and control flow*: our term dependence graph can be seen as a model for program dependences as those proposed in [16, 18]. An interesting feature of our proposal is its ability to naturally handle input-output system predicates.
- *Definition of relevance for argument positions*: we have defined a notion of relevance among the argument positions of a program. The notion of relevance is the essence of the slicing algorithm. An extra contribution is that within our framework it is possible to slice a logic program with respect to its side-effects.
- *Slicing algorithm*: its correctness, accuracy and costs hinge on the term dependence graph and the test for relevance among argument positions.

We would like to continue our work along the following lines of research:

- *Tutoring and Debugging*: the slicing algorithm can be embedded into a logic programming environment and used for debugging and tutoring [11]. We would like to assess the usefulness of such facility for learners of a logic programming language.
- *Program Reuse*: logic program slices can be made into partially developed programs (or *program techniques* [5]) for subsequent use in a program editor [25]. In particular, a slice can be converted into a program transformation schema, as in [34], and applied to existing programs, conferring extra functionalities on them.

Acknowledgements: The author would like to thank the anonymous referees for their helpful comments and Seumas Simpson for proofreading earlier versions of this text.

References

- [1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic Slicing in the Presence of Unconstrained Pointers. In *Proc. TAV4*. ACM/IEEE, Oct. 1993.
- [2] H. Ait-Kaci. *Warren's Abstract Machine*. MIT Press, USA, 1991.
- [3] K. R. Apt. *From Logic Programming to Prolog*. Prentice-Hall, U.K., 1997.
- [4] V. Berzins, editor. *Software Merging and Slicing*. IEEE Computer Society, 1995.
- [5] A. W. Bowles, D. Robertson, W. W. Vasconcelos, M. Vargas-Vera, and D. Bental. Applying Prolog Programming Techniques. *Int'l Journal of Human-Computer Studies*, 41(3):329–350, Sept. 1994.
- [6] J. Boye, J. Paaki, and J. Małuszyński. Synthesis of Directionality Information for Functional Logic Programs. In *Lect. Notes in Comp. Sci., Vol. 724*. Springer-Verlag, 1993.
- [7] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, 1987.
- [8] P. Cousout and R. Cousout. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.
- [9] S. K. Debray and D. S. Warren. Automatic Mode Inference for Prolog Programs. In *IEEE Symp. on Logic Progr.*, 1986.
- [10] R. A. DeMillo, H. Pan, and E. Spafford. Critical Slicing for Software Fault Localization. In *Proc. of ISSSTA'96*. ACM, 1996.
- [11] M. Ducassé and J. Noyé. Logic Programming Environments: Dynamic Program Analysis and Debugging. *Journal of Logic Programming*, 19, 20:351–384, 1994.
- [12] K. B. Gallagher and J. R. Lyle. Using Program Slicing in Software Maintenance. *IEEE Trans. on Soft. Eng.*, 17(8):751–761, Aug. 1991.
- [13] V. Gouranton. Deriving Analysers by Folding/Unfolding of Natural Semantics and a Case Study: Slicing. In *Proc. SAS'98, Lect. Notes in Comp. Sci., Vol. 1503*. Springer-Verlag, 1998.
- [14] V. Gouranton and D. Le Métayer. Dynamic Slicing: a Generic Analysis based on a Natural Semantics Format. RR 3375, INRIA, France, Mar. 1998.
- [15] T. Guymóthy and J. Paaki. Static Slicing of Logic Programs. In *Proc. 2nd. Int'l Workshop on Automated and Algorithmic Debugging*, IRISA, France, 1995.
- [16] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Trans. on Program. Lang. & Systems*, 12(1):26–60, Jan. 1990.
- [17] H. Huang, W. T. Tsai, and S. Subramanian. Generalized Program Slicing for Software Maintenance. In *Proc. SEKE'96, Nevada, U.S.A.*, 1996.

- [18] D. Jackson and E. J. Rollins. A New Model of Program Dependences for Reverse Engineering. In *Proc. SIGSOFT'94*. ACM, 1994.
- [19] T. Kanamori and T. Kawamura. Analyzing Success Patterns of Logic Programs by Abstract Interpretation. Technical Report 279, ICOT, June 1987.
- [20] B. Korel and J. Laski. Dynamic Program Slicing. *Inf. Proc. Letters*, 29(3):155–163, Oct. 1988.
- [21] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1993.
- [22] H. Mannila and E. Ukkonen. Flow Analysis of Prolog Programs. In *IEEE Symp. on Logic Progr.*, 1987.
- [23] C. Mellish. Abstract Interpretation of Prolog Programs. In *Abstract Interpretation of Declarative Languages*. Ellis Horwood, 1987.
- [24] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding Up Slicing. In *Proc. SIGSOFT'94*. ACM, 1994.
- [25] D. Robertson. A Simple Prolog Techniques Editor for Novice Users. In *3rd Annual Conf. on Logic Progr.* Springer-Verlag, 1991.
- [26] S. Schoenig and M. Ducassé. A Backward Slicing Algorithm for Prolog. In *Proc. SAS'96, Lect. Notes in Comp. Sci., Vol. 1145*. Springer-Verlag, 1996.
- [27] R. W. Sebesta. *Concepts of Programming Languages*. Addison-Wesley, 1996.
- [28] A. M. Sloane and J. Holdsworth. Beyond Traditional Program Slicing. In *Proc. of ISSSTA'96*. ACM, 1996.
- [29] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, 1986.
- [30] F. Tip. A Survey of Program Slicing Techniques. *Journal of Prog. Lang.*, 3(3):121–189, Sept. 1995.
- [31] W. W. Vasconcelos. A Method of Extracting Prolog Programming Techniques. Technical Paper 27, DAI, Edinburgh Univ., 1994.
- [32] W. W. Vasconcelos. *Extracting, Organising, Designing and Reusing Prolog Programming Techniques*. PhD thesis, DAI, Edinburgh Univ., Aug. 1995.
- [33] W. W. Vasconcelos and M. A. T. Aragão. Slicing Logic Programs. Tech. Report, 1998. Available from authors.
- [34] W. W. Vasconcelos and N. E. Fuchs. Prolog Program Development via Enhanced Schema-Based Transformations. RP 776, DAI, Edinburgh Univ., 1996.
- [35] M. Weiser. Programmers use Slices when Debugging. *Comm. of the ACM*, 25(7):446–452, 1982.
- [36] M. Weiser. Program Slicing. *IEEE Trans. on Soft. Eng.*, SE-10(4):352–357, 1984.

Applying Logic Programming to Derive Novel Functional Information of Genomes

Arvind K. Bansal¹★ and Peer Bork²

¹Department of Mathematics and Computer Science
Kent State University, Kent, OH 44242, USA
Phone: +1.330.672-4004 ext. 214 E-mail: arvind@mcs.kent.edu

²Computer Informatics Division,
European Molecular Biology Laboratory
Meyerhofstr 1, Postfach 10 22 09, 69012 Heidelberg, Germany
Phone: +49.6221.387.534 E-mail: Bork@EMBL-Heidelberg.de

Abstract. This paper describes an application of the logic programming paradigm to large-scale comparison of complete microbial genomes each containing four-million amino acid characters and approximately two thousand genes. We present algorithms and a Sicstus Prolog based implementation to model genome comparisons as bipartite graph matching to identify orthologs — genes across different genomes with the same function — and groups of orthologous genes — orthologous genes in close proximity, and gene duplications. The application is modular, and integrates logic programming with Unix-based programming and a Prolog based text-processing library developed during this project. The scheme has been successfully applied to compare eukaryotes such as yeast. The data generated by the software is being used by microbiologists and computational biologists to understand the regulation mechanisms and the metabolic pathways in microbial genomes.

Key-words. declarative programming, gene groups, genome comparison, logic programming, metabolic pathway, microbes, Prolog application, operons, orthologs

1 Introduction

Microbes serve as model organisms for understanding basic metabolic functions of living beings. Microbes are also important targets in biotechnology, disease treatment, and ecology. A natural step in understanding microbial genomes [1, 9] is to map the functionality and the variations in the functionality of genes and families of genes [19]. Knowledge of the functionality of genes will enhance our understanding of metabolic pathways — control flows of action among enzymes involved in high level process control in microbial organisms. Understanding the variations of metabolic pathways in two bacteria and their regulation is the key to the

† Corresponding author

★This research was supported in part by Research Council grant, Kent State University, Kent, Ohio, USA and Deutsche Forschungsgemeinschaft — The German Federal Agency

G. Gupta (Ed.): PADL'99, LNCS 1551, pp. 275-289, 1998.

© Springer-Verlag Berlin Heidelberg 1998

development of specific drugs to restrict the effects of harmful bacteria, and to enhance the effects of useful bacteria.

It is anticipated that the complete human genome will be fully sequenced by the year 2003. The development of mapping and comparison techniques will provide a basis to develop tools to understand human genome components: the functionality of genes and an understanding of metabolic pathways.

The genes inside a metabolic pathway are often clustered together in gene-groups called *operons*. The constituting genes of an operon are usually copied together (co-transcribed), and may have a common control region which facilitates their co-transcription. The identification of operons facilitates the identification of metabolic pathways using similarity analysis.

Currently, the functionality of many of the genes is available experimentally from wet laboratories. However, the function of genes and gene-groups of newly sequenced genomes is unavailable. Due to the size of genomes — an average microbial genome has 0.6 million – 4.7 million amino-acids and 472 – 4200 genes — and the increasing rate of availability of complete genomes, a cost-effective and time-efficient way is to use automated pair-wise comparison of genome sequences to identify functionally equivalent genes in newly sequenced genomes.

One technique to understand the functionality of genes in newly sequenced genomes is to identify *orthologs* — genes which have same function and a common evolutionary ancestor. However, It is difficult to ascertain common evolutionary ancestors due to the divergence and the convergence of species during evolution [8]. One heuristic is to identify two genes in evolutionarily close genomes [13, 18], represented as amino-acid sequences, with the highest similarity using pair-wise genome comparison. These most similar genes are the best candidates for being orthologs. We refer to such pair of genes as *putative orthologs*. These putative orthologs can be further confirmed by automated matching of secondary structure of amino-acid sequences and automated matching of the biochemical properties of different parts of the genes, and finally by wet lab techniques if needed.

Orthologous genes in close proximity are strong candidates to be involved together in a common function. Two gene-groups are *putative orthologous* — candidates for being involved together in a common function — if the constituting genes in the gene-groups are putative orthologs. These putative orthologous gene-groups may include one or more operons, and are natural starting points to identify operons and metabolic pathways.

The first step is to identify *homologs* — similar sequences identified using approximate string matching. Homologs include both *putative orthologs* and *paralogs* — duplicated genes with different functions [8]. Paralogs are pruned out from homologs leaving putative orthologs.

In this paper, we use previously developed software to identify [3, 11] and align homologs [12, 16, 21], and develop graph matching techniques to identify putative orthologs, homologous gene-groups, duplicated gene-groups, and putative orthologous gene-groups. We have modeled each genome as an ordered set of genes, modeled the matching of a pair of complete genomes as a bipartite graph matching problem, developed algorithms [7] to identify homologous gene-groups, and developed a variant of the stable marriage algorithm [14] to identify pairs of genes

with the highest similarity scores. The bipartite graph has 2000 to 4000 nodes in each set and approximately 12000 edges.

The problem required the integration of text processing, approximate string matching and dynamic programming techniques for gene comparison (already available in C), as well as various techniques such as heuristic reasoning, high level modeling using graphs and trees, and a high level interface to system programming. The problem is suitable for logic programming because:

1. The software must be experimental and continuously evolving since the output data are continuously providing new insights into the genomic structure and function. Many pieces of information [7] previously unknown are being identified and investigated by the microbiology community [20].
2. Prolog facilitates easy integration of heuristic reasoning, text processing and manipulation of complex data structures such as graphs and associative tables.

Large scale iteration has been simulated using tail recursive programming. The tool was developed using Sicstus Prolog version 3.5 which has a library to interface with Unix-based systems and a library to manipulate graphs and associations. Currently the software takes around 180 minutes to compare and derive information for a pair of microbial genomes — each having approximately 4000 genes and around four million amino-acid characters — on a four processor SGI with 195 Mhz processors. The majority of time is taken by BLAST comparisons [3, 11] and the Smith-Waterman gene alignment [12, 16, 21]) to align the homolog-pairs. The detection of functional information about the genome (described in this paper) takes less than 100 seconds of execution for a bipartite graph consisting of approximately 4200 nodes in each set and approximately 12000 edges.

The claim of this paper is that this state-of-the-art Prolog software has automated the innovative analysis of genome-data to identify the functionality of micro-organisms [7]. The software identifies gene-groups, orthologs, gene-fusions — two genes fusing together to form a new gene, gene-duplications — single genes having multiple occurrences in another genome, and gene-group duplications — groups of genes in one genome having multiple occurrences in another genome. Such detailed information was not previously available to the biological community. The data generated by an extended version of this software¹ is being used by researchers to study gene regulation [20], and derive metabolic pathways [6]. The software executes in a realistic time for a computationally intensive problem and can be modified easily as new information about genome organization is discovered.

The paper is organized as follows: Section 2 describes the concepts and related software; Section 3 describes the overall scheme; Section 4 describes the integration of Unix-based programming and text processing to extract gene-related information from Genbank at NCBI, invocation of Unix shells for BLAST comparisons, and invocation of Unix shells for gene-pair alignment; Section 5 briefly describes the algorithm [7] for identifying homologous gene-groups and the related Prolog implementation; Section 6 describes an algorithm for identifying orthologs and the related Prolog implementation; and Section 7 briefly describes the related works, and the last section concludes the paper.

¹ The orthologs were confirmed by a variation of the Hungarian method for bipartite graph matching developed by Peter Stuckey [7].

2 Background

The author assumes a basic familiarity with Prolog [2, 17]. In this section, we describe some genome related concepts [1] needed to understand the paper, and a brief description of bipartite graphs needed to model genome comparison.

2.1 Genome Related Background

The genome of an organism is encoded within molecules of DNA [1, 16]. A molecule of DNA is a sequence composed of four nucleotides represented as ‘A’, ‘C’, ‘G’, ‘T’. A protein is a sequence of twenty amino acids represented as alphabets of English language.

A *microbial genome* is modeled as an ordered set of pairs of the form $\langle (c_1, \gamma_1), \dots, (c_N, \gamma_N) \rangle$. Each γ_i is a sequence of the form $\langle s_1, \dots, s_N \rangle$ ($1 \leq i \leq N$) where s_i is a nucleotide for DNA and an amino-acid for a protein. Each c_i is a control region preceding γ_i . In this paper, we are interested in the comparison of protein sequences of the form $\langle \gamma_{11}, \dots, \gamma_{1N} \rangle$ and $\langle \gamma_{21}, \dots, \gamma_{2M} \rangle$. In this paper, we will denote a complete genome by the subscripted Greek letter Γ_i and individual gene by γ_i and a gene-pair by $(\gamma_{1i}, \gamma_{2i})$ where the first subscript denotes the genome number, and the second subscript denotes the sequential order of a gene within a genome.

Two gene-sequences are similar, if there is a significant match between them after limited shifting of characters. *Sequence alignment* arranges similar sequences in a way that asserts a correspondence between characters that are thought to derive from a common ancestral sequence. Aligning a set of sequences requires the introduction of spacing characters referred to as *indels*. If the aligned sequences did indeed derive from a common ancestral sequence, then indels represent possible evolutionary events. While aligning two sequences, a two dimensional matrix (PAM matrix) describing similarity between amino-acids is used [16, 21]: amino-acids with similar biochemical properties are better suited for identifying the similarity of a protein.

Example 1.1 Let us consider two amino-acid sequences ‘AGPIAL’ and ‘APVV’. The amino-acids ‘I’, ‘L’, and ‘V’ are very similar and belong to a common category: hydrophobic amino-acids. Thus matching the amino-acid ‘I’ with ‘V’ or matching the amino-acid ‘I’ with ‘V’ is preferred. A possible alignment is given below:

```

A G P I A L
A _ P V _ V

```

2.1 Sequence Comparison Techniques

The BLAST software [3, 11] — a popular technique to search for homologs — selects a small subsequence, using string matching to identify locations of matches, and expands the size of matched segments at the identified locations using finite state automata and approximate character matching using a PAM matrix [9].

The Smith-Waterman algorithm [12, 16, 21] is a matrix-based dynamic programming technique to align two sequences. The Smith-Waterman alignment is based upon iterative addition of the next position to a pair of best matching subsequences. There are three possibilities: the next position in the first sequence

matches an indel, the next position in the second sequence matches an indel, or two non-indel characters match. The algorithm uses the function:

$$\text{similarity_score}(A[1..I], B[1..J]) = \max(\begin{aligned} &\text{similarity_score}(A[1..I-1], B[1..J]) + \text{penalty}(a_i, \text{'_'}), \\ &\text{similarity_score}(A[1..I], B[1..J-1]) + \text{penalty}(\text{'_'}, b_j), \\ &\text{similarity_score}(A[1..I-1], B[1..J-1]) + \text{match}(a_i, b_j) \end{aligned})$$

where $A[1..K]$ or $B[1..K]$ is a subsequence of the length K starting from the beginning position, and a_i or b_j is an element in the sequence. The Smith-Waterman algorithm is more precise than BLAST. However, BLAST is more efficient than the Smith-Waterman algorithm.

2.2 Functional Genomics Related Concepts

Homologs are similar genes derived from similarity search techniques such as BLAST [3, 11]. *Paralogs* are homologous genes resulting from gene duplication, and have variations in functionality. A *putative ortholog* is a homolog with the highest similarity in pair-wise comparison of genomes.

A *gene-group* is a cluster of neighboring genes $\langle \gamma_{I1} \gamma_{IJ} \gamma_{IK} \dots \rangle$ ($I < J < I + r$, $J < K < J + r$, where $r > 0$). A gene-group may have insertions, permutations, or deletions of genes with reference to a corresponding gene-group in another genome. A *homologous gene-group* $\langle \gamma_{I1} \gamma_{IJ} \gamma_{IK} \dots \rangle$ ($I < J < K$) in the genome Γ_1 matches with the corresponding gene-group $\langle \gamma_{2M} \gamma_{2N} \gamma_{2P} \dots \rangle$ ($M < N < P$) in Γ_2 such that γ_{I1} and γ_{IJ} and γ_{IK} etc. are similar to one of the genes in the sequence $\langle \gamma_{2M} \gamma_{2N} \gamma_{2P} \dots \rangle$. A *shuffled gene* is an ortholog of a gene in a gene-group such that the ortholog lies outside the putative orthologous gene-group. *Gene-gaps* are genes in one genome without any orthologous correspondence in another genome. A *fused gene* in a genome has two orthologs which fuse together to form a single gene.

A genome is modeled as an *ordered set of genes*. Pair-wise genome comparison is modeled as a weighted bipartite graph such that genes are modeled as vertices, two homologous genes in different sets have edges between them, and the similarity score between two genes is the weight of the corresponding edge.

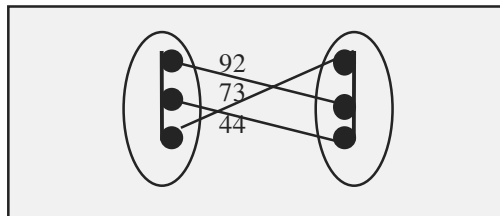


Fig. 1. Genome comparison as bipartite graph matching problem

3 A Scheme to Identify Putative Orthologs and Gene-Groups

In this section, we describe an overall scheme for identifying functionally equivalent genes and gene groups. The identification involves six stages: a protein preparation stage, a homolog detection stage, an alignment stage, a gene-group identification stage, and an ortholog detection stage. The output after each stage is in the form of a text file. The advantages of representing the information in readable textual form are:

1. The development of each stage is modular and independent of others.
2. The output after each stage is in human comprehensible form for further analysis.

The software development is incremental, and is dependent upon the feedback of microbiologists and computational biologists after each stage. The textual form of representation facilitates this development.

Some major features of the text-processing software are: skipping white characters, removing white characters, skipping text until a marker text is identified, splitting a line into various words, splitting an annotated word into smaller information units, splitting a string into two substrings after identifying a pattern, and converting strings into numbers or atoms and *vica-versa*.

The first stage prepares the data for comparison of two genomes. This stage extracts protein sequences from annotated text files given in Genbank (<ftp://ncbi.nlm.nih.gov/genbank/genomes/bacterial/>) and requires text-processing capability (coded in Sicstus Prolog). The software extracts the information about the *name of a gene, starting point and ending point of the protein-coding region, length of a gene, and the DNA-strand* used to extract the amino-acid sequence of the gene.

The second stage uses BLAST [11] to identify a set of homolog-pairs. The output of this stage is of the form $(\gamma_{11}, \gamma_{21}, \textit{similarity-score})$: gene γ_{11} occurs in the first genome, the gene γ_{21} is a homolog of γ_{11} in the second genome, and *similarity-score* describes the amount of similarity between two genes. This method improves the execution efficiency by pruning edges with low similarity score.

The third stage aligns two homologous genes using the Smith-Waterman alignment software [12]. The output of this stage is of the form $(\gamma_{11}, \gamma_{21}, \textit{alignment-score}, \textit{identity-score}, \textit{beginning shift}, \textit{end-shift}, \textit{size of largest segment of continuous indels})$.

The fourth stage models the pair of genomes as a weighted bipartite graph with each genome forming an ordered set. Putative groups of homologous genes are identified by traversing all the genes in the genomes and identifying the clusters of edges whose adjacent vertices in both the sets in the bipartite graph are within a predetermined range. As the new vertices are selected the range keeps expanding since it is centered around the current vertex. The gene-group breaks when there is at least one source-vertex which has all the edges ending in sink vertices out of the proximity of the previously matching vertices.

The fifth stage uses the bipartite graph to identify the best possible edge connecting two nodes. These nodes with best matches are marked as orthologs. Empirical data and the study of enzyme nomenclatures, other secondary structure prediction methods [20], and other bipartite graph matching techniques [7] have suggested that the scheme is a good heuristic for identifying the putative orthologs.

The sixth stage identifies the orthologous gene-groups by combining the knowledge of homologous gene groups and orthologs. Many of these groups may be operons or a group of operons involved in metabolic pathways. This stage also identifies shuffled genes, gene-gaps, and fused genes.

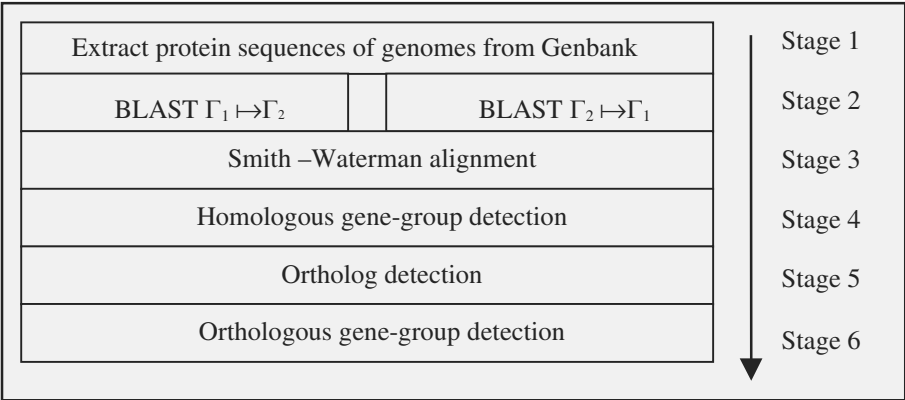


Fig 2. A complete schematics to identify orthologs and orthologous gene-groups

4 **Implementing Stages I, II, and III**

In this section, we briefly describe the implementations of Stage I — extracting genome information from Genbank, Stage II — identifying homologous genes using BLAST [11], and Stage III — alignment of homologous gene-pairs using the Smith-Waterman software [16].

4.1 **Stage 1: Extracting Genome Information from Genbank**

The amino acid sequences given in the Genbank (*ftp://ncbi.nlm.nih.gov/genbank/genomes/ bacterial*) in the GBK format are extracted and processed using the text-processing library developed in Prolog. The software generates two text files: a file containing an ordered set of protein coding regions of genes in FASTA format, and a file containing the ordered set of 5-tuples (*gene-name, index of the gene, location of first nucleotide of a gene, location of the last nucleotide of a gene, the direction of the gene*). The information in the second file is later used to identify and analyze the control regions of orthologs.

4.2 **Stage II: Identifying Homologs by BLAST Comparisons**

In the case of the BLAST search, one of the genomes is converted as a database of genes, and every gene in the second genome is matched (using a Bourne shell) against

the database for similarity. The code to invoke BLAST comparison is illustrated in Figure 3.

The predicate *homologs/4* takes one of the genomes as an input stream *GeneStrm*, and the other genome as a database *Database* created by a special blast command. The predicate *homologs/4* tail-recursively extracts the next gene sequence in a temporary file *Gene* using the predicate *next_gene/2*, invokes the predicate *blastp/5* to identify homolog-pairs, and deletes the temporary file *Gene* after collecting the homologs for the gene.

```
homologs(GeneStream, Database, Threshold, OutStream):-
    (next_gene(GeneStream, Gene) ->
        blastp(Gene, Database, Threshold, OutStream),
        homologs(GeneStream, Database, Threshold, OutStream)
        delete_file(Gene)
    ;otherwise -> true).

blastp(Gene, Database, Threshold, OutStream):-
    temp_file(Result),
    shell_cmd([blastp, Gene, DataBase, '>', Result]),
    filter_high_pairs(Result, Threshold, OutStream),
    delete_file(Result).
```

Fig. 2. Identifying homologs by invoking BLAST using Bourne shell

The predicate *blastp/5* creates a temporary file *Result*, invokes a Bourne shell using the predicate *shell_cmd/1* which writes the matching gene-pairs with similarity scores in the temporary file *Result*, filters out the gene-pairs above a user-defined threshold value *Threshold* using the procedure *filter_high_pairs/3*, and finally deletes the file *Result*.

The predicate *shell_cmd/1* fuses the elements of the list into one Unix command, and uses the built-in predicate *shell/1* to invoke the command '*blastp GeneFile DataBase > Result*'. The predicate *temp_file/1* creates a random file previously absent in the current subdirectory.

4.3 Stage III: Aligning Homolog-Pairs

For the gene-alignment stage, every homolog-pair is aligned using separate Bourne shells. The shell invocation is given in Figure 4.

The predicate *align_genome/4* takes as inputs *HomologStrm* — the stream of homolog-pairs produced by the BLAST comparisons, *HomologAssoc* — an association table of attributes of homolog-pairs, *Parameter* — a set of parameters needed by the alignment command *mlocalS* [16], and outputs a similarity score and other alignment related attributes. The predicate *next_homolog_pair/2* extracts the next homolog-pair from the stream *HomologStrm*. The predicate *get_assoc/3* extracts the homolog-related attributes *HomologAttrs* from the association table *HomologAssoc*. The predicate *alignment/3* produces alignment related attributes *AlignAttrs* of a homolog-pair. The predicate *append/2* appends the gene names and homolog related attributes and alignment related attributes. The predicate *write_score/2* writes the information in the output stream *OutStrm*.

The predicate *alignment/3* accepts a homolog-pair (*Gene1*, *Gene2*) and a set of parameters *Parameter*, and outputs a similarity score in the file *ScoreFile* after the alignment of the homolog-pair. The predicate *temp_file/1* creates a temporary file *ScoreFile* where the similarity score is written by the Smith-Waterman software in a textual form. The predicate *similarity_score/4* extracts the similarity score and other alignment-related attributes from the file *ScoreFile* using text processing. The predicate *delete_file/1* deletes the file *ScoreFile*.

The Smith-Waterman software [12] invokes a command '*mlocalS GeneFile1 GeneFile2 Matrix GapPenalty MultipleGapPenalty > Scorefile*'. The gene sequences are accepted in files *GeneFile1* and *GeneFile2*, and similarity score and other alignment related attributes are written in the file *ScoreFile*. The names of the files created for each gene is the same as gene-names. The predicate *absolute_file/2* takes a gene name, and returns an absolute file.

```
align_genome(HomologStrm, HomologAssoc, Parameter, OutStrm):-
    (next_homolog_pair(HomologStrm, Source-Sink) ->
     get_assoc(Source, HomologAssoc, HomologAttrs),
     alignment((Source, Sink), Parameter, AlignAttrs),
     append([[Source, Sink], HomologAttrs, AlignAttrs], Line),
     write_scores(OutStrm, Line),
     align_genome(HomologStrm, HomologAssoc, Parameter, OutStrm)
    ;otherwise -> true).

alignment(HomologPair, Parameter, AlignAttrs):-
    HomologPair = (Gene1, Gene2),
    temp_file(ScoreFile),
    align_gene_pair(Gene1, Gene2, Parameter, ScoreFile),
    similarity_score(ScoreFile, AlignAttrs),
    delete_file(ScoreFile).

align_gene_pair(GeneName1, GeneName2, Parameter, OutFile):-
    Parameter = (Matrix, Offset, GapPenalty, MultiGapPenalty),
    absolute_file(GeneName1, GeneFile1),
    absolute_file(GeneName2, GeneFile2),
    shell_cmdnd([mlocalS, GeneFile1, GeneFile2, Matrix, Offset,
                 GapPenalty, MultiGapPenalty, '>', OutFile])).
```

Fig. 3. Code to invoke a shell for the Smith-Waterman gene-pair alignment

5 Identification of Homologous Gene-Groups

In this section, we describe an algorithm to identify homologous gene-groups in two genomes, and a Prolog implementation for the corresponding algorithm. This algorithm is based upon identifying islands of edges such that sets of sources and sinks in the bipartite graph are in proximity in the corresponding ordered sets. The implementation models the bipartite graph, and then identifies the islands of edges in proximity.

5.1 Modeling Genome Comparison as a Bipartite Graph Matching Problem

A weighted bipartite graph is modeled as an associative table of pairs of the form (source-node, value). The use of an associative table facilitates random access of the

source genes and the corresponding sinks and the related attributes during the bipartite graph matching. The value is a list of 5-tuples of the form (*sink node, weight of the edge given by the similarity score, gene-strand information, length of the gene, length of the aligned portion of the gene*).

The code for modeling the pair-wise genome comparison as bipartite graph is given in Figure 5. The predicate *edges/2* collects all the gene-pairs from the output of the gene alignment stage. The predicate *source_vertices/2* collects all the source nodes from the gene-pairs. The predicate *group_edges_by_source/3* groups the edges as a list of pairs (*source-node, the list of edges and related attributes*), and forms an association out of the list to facilitate efficient access of the edges and their attributes.

```
bipartite_graph(Stream, BipartiteAssoc) :-
    edges(Stream, Edges), source_vertices(Edges, Nodes),
    group_edges_by_source(Nodes, Edges, Graph),
    list_to_assoc(Graph, BipartiteAssoc),
    group_edges_by_source([ ], _, [ ]).

group_edges_by_source([Src|Ss], Edges, Graph) :-
    group_edges_for_node(Edges, Src, RemainingEdges, SubGraph),
    Graph = [Src-SubGraph|Gs],
    group_edges_by_source(Ss, RemainingEdges, Gs).

group_edges_for_node([ ], _, [ ], [ ]).
group_edges_for_node([Source-SinkAttributes|Es], Node,
    RemainingEdges, SubGraph) :-
    (Source == Node ->
        SubGraph = [SinkAttributes|Ss],
        group_edges_for_node(Es, Node, RemainingEdges, Ss)
    ;otherwise ->
        RemainingEdges = [Source-SinkAttributes|Es],
        SubGraph = [ ]).

edges(Stream, ListofEdges) :-
    skip_white_spaces(Stream, NextChar),
    is_end_of_file(NextChar) -> ListofEdges = [ ]
    ;otherwise -> next_line(Stream, Line),
        weighted_edge(Line, WeightedEdge),
        ListofEdges = [WeightedEdge|Es],
        edges(Stream, Es).
```

Fig. 4. Modeling genome comparison as bipartite graph matching

5.2 Identifying Islands of Clustered Edges for Homologous Gene-Groups

In the following algorithm, we denote a set of elements in the I_{th} field in a set of m-tuples S by $\Pi_i(S)$. Given a set of gene-pairs S , the set of source-vertices will be denoted as $\Pi_1(S)$, and the set of sink vertices will be denoted as $\Pi_2(S)$.

The technique uses a sliding window of size b ($b \geq 1$) to identify the cluster of edges in close proximity. For any edge $(\gamma_{1l}, \gamma_{2l})$, the process is repeated if another matching $(\gamma_{1(l+r)}, \gamma_{2(l \pm s)})$ ($0 < r, s < b$) is found. The next search is performed in the range $(J - b, J + s + b)$ for the matching $(\gamma_{1(l+r)}, \gamma_{2(l+s)})$, in the range $(J - s - b, J + b)$ for the matching $(\gamma_{1(l+r)}, \gamma_{2(l-s)})$. Since the window keeps sliding, variably-sized gene-groups are identified.

Let the set of nodes in Γ_1 (or Γ_2) be S_1 , and the set of unprocessed weighted edges be S_2 . Every time an edge $(\gamma_{1l}, \gamma_{2l}) \in$ set of remaining edges is picked such that γ_{1l} has the least value of l . S_3 — the neighborhood set of γ_{1l} — is $\{\gamma_{1(l-b)}, \dots, \gamma_{1(l+b)}\}$ such that $0 \leq l < b$ and $l + b \leq \text{number of genes in } \Gamma_1$. Let the set of all the nodes in Γ_2 which match with γ_{1l} be S_4 . Let the set of nodes in S_4 which share edges with the nodes in the neighborhood set S_3 be S_5 . The set S_5 is identified by selecting the nodes in S_4 , one at a time, and taking the intersection of the set of nodes in Γ_1 connected to the node and the set S_3 . If the intersection is non-empty then that node in S_4 is included in S_5 . If the set S_5 is non-empty then there is at least one possibly matching gene-group. After detecting the presence of a homologous gene-group, the nodes in Γ_1 are traversed from the node γ_{1l} , and the putative gene-groups are collected.

To facilitate the dynamic alteration of neighboring nodes, a copy of the sets S_3 and S_5 is made in the sets S_7 and S_9 respectively. A set S_8 is used to verify matching edges incident upon the nodes in S_3 during the collection of variably sized gene-groups. If there is a matching node within the range $\gamma_{1(K+r)} (0 \leq r < b, K < l) \in S_7$ which matches one of the nodes $\gamma_{2L} \in S_9$, then there is a group. The set of edges in a neighborhood is collected in S_6 ; S_7 — the neighborhood set of $\gamma_{1K} (K \geq l) \in S_7$ — is extended dynamically to include the neighbors of $\gamma_{1(K+r)}$; S_9 — the neighborhood set of matching nodes in Γ_2 — is extended to include the neighbors of γ_{2L} ; and all the edges incident on $\gamma_{1(K+r)} (0 \leq r < b \text{ and } K \geq l)$ are included dynamically in the set S_8 . Those edges which are traversed once are deleted from S_8 , and those nodes in Γ_2 which have been traversed once are deleted from the sets S_5 and S_9 . After a matching $(\gamma_{1(K+r)} \in S_7) \mapsto (\gamma_{2L} \in S_9)$ is not found in the neighborhood, S_6 — the current collected set of edges — is closed; $\Pi_1(S_6)$ — the set of source-nodes in S_6 — gives a putative corresponding gene-group in Γ_1 , and $\Pi_2(S_6)$ — the set of sink-nodes in S_6 — gives the putative homologous gene-group in Γ_2 . The set of nodes in Γ_2 which have been traversed during the identification of the last group is deleted from the set S_9 , and the set of traversed edges is deleted from the set S_8 .

The process is repeated to identify the next homologous gene-group which starts with γ_{1l} . This is possible since there are duplicates of gene-groups. After finding out all the homologous gene-groups involving γ_{1l} , the set of edges incident upon γ_{1l} is deleted to avoid reconsideration. The process is repeated by picking up the next edge in S_2 which has the minimum index, until S_2 is empty.

5.3 Implementing the Algorithm

Figure 6 illustrates the top level code for the implementation of the algorithm. The predicate *gene_groups/3* picks up all the edges starting from a gene in the variable *EdgesfromSource*, extracts the corresponding sink vertices using the predicate *sink_vertices/2*, and identifies the list of genes in the proximity which have neighbors using the predicate *have_neighbors/4*, and then picks up all the groups associated with the neighboring genes *HavNbrs* using the predicate *islands/6*.

The predicate *island/6* picks up the groups *Groups* corresponding to one source node *Node* using the predicate *one_group/5*, deletes the corresponding edges incident upon the source node *Node* from the graph *Graph* using the predicate *delete_edges/3*,

and repeats the process with the remaining nodes *G_s* and the remaining subgraph *Rest*.

```
gene_groups([ ], _, [ ]).
gene_groups(Graph, Boundary, GeneGroups):-
    Graph = [EdgesfromSource|Gs],
    EdgesfromSource = Source-SinksandAttributes,
    sink_vertices(SinksandAttributes, Sinks),
    (have_neighbors(Source-Sinks, Boundary, Graph, HavNbrs)->
        Prox is Boundary - 1,
        islands(HavNbrs, Boundary, Prox, Graph, Group, Rest),
        GeneGroups = [Group|Rs],
        gene_groups(Rest, Boundary, Rs)
    ;otherwise -> gene_groups(Gs, Boundary, GeneGroups)).

islands([ ], _, _, Graph, [ ], Graph).
islands([Node|Gs], Boundary, Prox, Graph, Groups, New):-
    one_group(Node, Boundary, Prox, Graph, Group),
    delete_edges(Group, Graph, Rest),
    (has_more_than_two_edges(Group) ->
        Groups = [Group|Ns],
        islands(Gs, Boundary, Prox, Rest, Ns, New)
    ;otherwise ->
        islands(Gs, Boundary, Prox, Rest, Groups, New)).
```

Fig. 5. Top level code to identify homologous gene-groups

6 Identification of Putative Orthologs

The technique sorts edges in the descending order of weight, and marks the node-pairs with the highest weights as putative orthologs, and all the edges incident upon the identified putative orthologs are pruned. The process is repeated until gene-pairs are consumed. The genes inside homologous gene-groups are positively biased since genes inside a gene-group are more likely to retain the function of an operon. In case there are two edges from a node with a very high score (above a threshold value), both are treated as putative orthologs. If there are two edges whose similarity score is below the threshold and both the scores are close, then both homologs are marked as conflicting orthologs. The conflicting orthologs are resolved by structure analysis or metabolic path analysis, and are outside the scope of this paper.

6.1 Implementing the Ortholog Detection

The top-level code for ortholog detection is given in Figure 7. The predicate *ortholog/3* accepts a list of gene groups in the variable *GeneGroups*, the bipartite graph association in the variable *BipartiteAssoc*, and derives the orthologs in the variable *Orthologs*. The homologous genes inside the gene-groups are positively biased by a bias factor *Factor* using the predicate *bias_grouped_edges/3*. The predicate *biased_edges/3* picks up non-grouped weighted edges from the association *BipartiteAssoc*, and picks up the weighted biased edges from the list *BiasedEdges*. The joint list of the weighted edges *WeightedEdges* is sorted in the descending order by weight using the predicate *descending_sort/2*. The resulting list of edges

SortedEdges is processed by the predicate *stable_marriage/2* to derive the homolog-pairs with the highest similarity.

The predicate *stable_marriage/2* invokes predicates *best_match/4* and *filter_well_separated/4*. The predicate *best_match/4* identifies edges with the highest weight, checks whether any previously processed edge already used any node in the current node-pair, and checks whether the difference in the weights of the previously processed edge and the current edge is significant. The predicate *filter_well_separated/2* filters out the clean orthologs.

```
orthologs(GeneGroups, BipartiteAssoc, Orthologs):-
    flatten_gene_groups(GeneGroups, GroupedHomologs),
    '$bias_factor'(Factor),
    bias_grouped_edges(GroupedHomologs, Factor, BiasedEdges),
    biased_edges(BiasedEdges, BipartiteAssoc, WEdges),
    descending_sort(WEdges, SortedEdges),
    stable_marriage(SortedEdges, UnsortedOrthologs),
    sort_by_gene_index(UnsortedOrthologs, Orthologs).

stable_marriage(Edges, Orthologs):-
    list_to_assoc([ ], SrcAssoc),
    list_to_assoc([ ], SinkAssoc),
    best_match(Edges, SrcAssoc, SinkAssoc, MarkedPairs),
    filter_well_separated(MarkedPairs, Orthologs).

best_match([ ], SrcAssoc, _, SrcAssoc).
best_match([WEdge|Cs], SrcAssoc, SinkAssoc, Marker):-
    WEdge = Weight-Src-Sink-SrcSt-SrcEnd-SinkSt-SinkEnd,
    get_value(Src-SrcSt-SrcEnd, SrcAssoc, Weight1),
    get_value(Sink-SinkSt-SinkEnd, SinkAssoc, Weight2),
    ((are_separated(Weight, Weight1),
      are_separated(Weight, Weight2)) ->
      I1 = Sink-Weight-SrcSt-SrcEnd-SinkSt-SinkEnd-separated
      I2 = Src-Weight-SinkSt-SinkEnd-SrcSt-SrcEnd-separated
      insert_first(Src, I1, SrcAssoc, NewSrcAssoc),
      insert_first(Sink, I2, SinkAssoc, NewSinkAssoc)
    ;otherwise ->
      I1 = Sink-Weight-SrcSt-SrcEnd-SinkSt-SinkEnd-conflict,
      I2 = Src-Weight-SinkSt-SinkEnd-SrcSt-SrcEnd-conflict,
      insert_last(Src, I1, SrcAssoc, NewSrcAssoc),
      insert_last(Sink, I2, SinkAssoc, NewSinkAssoc)),
    best_match(Cs, NewSrcAssoc, NewSinkAssoc, Marker).
```

Fig. 6. Top level code to identify orthologs

Two association tables are kept in the predicate *best_match/4* to check the presence of a previously processed edge and to compare the weights of previously processed edge and the current edge. The first association *SourceAssoc* stores a double-ended queue as the value of the source-vertex in the bipartite graph. The second association table *SinkAssoc* stores a double ended queue as the value of the sink-vertex in the bipartite graph. If the difference in the weights of the previously processed edge and the current edge is significant then the current edge is discarded. First edge is inserted in front of the associations *SourceAssoc* and *SinkAssoc*. If the difference in the weight of previously processed edge and the current edge is insignificant, then both edges are marked *conflicting*. Conflicting edges are resolved by biological reasoning. The predicate *get_value/3* gets the weight from the association for comparison with

the current weight. The predicate *are_separated/2* checks whether the difference between weights of the previously processed edge and the current edge is significant. The predicate *insert_last/4* inserts an edge and its attributes at the end of the deque. The predicate *insert_first/4* inserts an edge and its attributes in the front of the deque.

7 Related Works

There are two types of related works: ortholog detection [19] and application of logic programming to genome related problems.

The work on ortholog detection [19], done independently and concurrently to our work, identifies clusters of orthologs using pair-wise comparison of all the proteins in a limited set of multiple genomes using *semi-automated* software and techniques. Our contribution over this work is that we have developed algorithms and implemented software for the completely automated identification of orthologs, gene-groups, shuffled genes. In addition, we are also able to identify gene fusions and duplicated gene-groups.

Logic programming has been applied to develop logical databases to retrieve information about metabolic pathways [10], to identify and model genome structure [4, 15], and to identify constrained portion of genes by integrating the information of phylogenetic tree and multiple sequence alignments [5]. All these works are significant for different aspects of genome modeling and analysis. However, this application performs large scale comparison of complete genomes to identify gene functions and other novel information significant for automated identification of operons and function of genes in newly sequenced genomes.

8 Future Works and Conclusion

We have described a novel application of logic programming to derive new functional information about microbial genomes. Logic programming is suited since the software has to be continuously modified to incorporate new changes based upon the analysis of the previous output. Logic programming integrates string processing, text processing, system-based programming, and heuristic reasoning. Currently the software is being modified to understand the constituent domains within genes, regulation mechanisms within genes [20], and to derive the metabolic pathways automatically [6].

Acknowledgements

The first author acknowledges lively discussions with the colleagues at EMBL, especially with Chris Sander and his group during Fall 1995. Warren Gish provided the latest version of BLAST [11]. Paul Hardy provided the portable library of the Smith-Waterman alignment software [12]. Peter Stuckey confirmed the results using a variant of the Hungarian method [7].

References

- [1] Alberts, B., Bray, D., Lewis, J., Raff, M., Roberts, K., and Watson, J. D.: Molecular Biology of THE CELL, Garland Publishing Inc. 1983
- [2] Almgren, J., Anderson J., Anderson S., et. al.: Sicstus 3 Prolog Manual. Swedish Institute of Computer Science, 1995
- [3] Altschul, S. F., Gish, W., Miller, W., Myers, E. W., Lipman, D. J: Basic Alignment Search Tools, J. Mol. Biol., vol. 215, (1991) 403 - 410
- [4] Baby, O. and : Non-Deterministic, Constraint-Based Parsing of Human Gene, Ph. D. thesis, Brandeis University, USA, (<http://www.cs.brandeis.edu/~obaby/phdthesis.html>)
- [5] Bansal, A. K.: Establishing a Framework for Comparative Analysis of Genome Sequences. Proceedings of the International Symposium of Intelligence in Neural and Biological Systems, Herndon VA USA (1995) 84 – 91
- [6] Bansal, A. K.: Automated Reconstruction of Metabolic Pathway of Newly Sequenced Microbial Genomes using Ortholog Analysis. to be submitted
- [7] Bansal, A. K., Bork, P., and Stuckey P.: Automated Pair-wise Comparisons of Microbial Genomes. Mathematical Modeling and Scientific Computing, Vol. 9 Issue 1 (1998) 1 – 23
- [8] Fitch, W. M.: Distinguishing Homologous from Analogous Proteins. Systematic Zoology, (1970) 99 - 113
- [9] Fleischmann, R. D., Adams, M. D., White O., et. al.: Whole-Genome Random Sequencing and Assembly of Haemophilus influenzae Rd. Science 269 (1995) 496 - 512
- [10] Gaasterland, T., Maltsev, N., and Overbeek, R.: The Role of Integrated Databases In Microbial Genome Sequence Analysis and Metabolic Reconstruction. In: Proceedings of the Second International Meeting on Integration of Molecular Biological Databases, Cambridge, England, July 1995.
- [11] Gish, W.: WU-BLAST version 2.0. Washington University, St. Louis MO USA, (<http://blast.wustl.edu>)
- [12] Hardy, P. and Waterman, M. S.: The Sequence Alignment Software Library. University of Southern California LA USA, (<http://www-hto.usc.edu/software/seqaln/>)
- [13] Olsen, J., Woese, C. R., and Overbeek R.: The Winds of Evolutionary Change: Breathing New Life into Microbiology. Journal of Bacteriology, Vol. 176 issue 1 (1994) 1 - 6
- [14] Papadimitrou, C. H., and Steiglitz, K.: Combinatorial Optimization: Algorithm and Complexity. Prentice Hall, (1982)
- [15] Searls, D. B.: The Linguistics of DNA. American Scientist 80: 579 - 591
- [16] Setubal, J. and Meidanis J.: Introduction to Computational Biology. PWS Publishing Company, (1997)
- [17] Sterling, L. S.. and Shapiro, E. Y.: The Art of Prolog. MIT Press, (1994)
- [18] Tatusov, R. L., Mushegian, M., Bork P. et. al.: Metabolism and Evolution of Haemophilus Influenzae Deduced From a Whole-Genome Comparison with Escherichia Coli. Current Biology, Vol. 6 issue 3 (1996) 279 - 291
- [19] Tatusov, R. L., Koonin, E. V., Lipman, D. J.: A Genomic Perspective on Protein Families. Science, Vol. 278 (1997) 631 - 637
- [20] Vitreschak, A., Bansal, A. K., Gelfand, M. S.: Conserved RNA structures regulation initiation of translation of Escherichia coli and Haemophilus influenzae ribosomal protein operons. First International Conference on Bioinformatics of Genome Regulation and Structure, Novosibirsk, Russia, (August 1998) 229
- [21] Waterman, M. S.: Introduction to Computational Biology: Maps, Sequences, and Genomes. Chapman & Hall, (1995)

An Application of Action Theory to the Space Shuttle

Richard Watson

Department of Computer Science
University of Texas at El Paso
El Paso, Texas 79968, U.S.A.
`rwatson@cs.utep.edu`

Abstract. We present a domain description for verifying plans concerning the Reaction Control System (RCS) of the Space Shuttle. Our approach utilizes recent research in action description languages. We describe the syntax and semantic of the action language, \mathcal{L}_0 , as well as a general translation from domain descriptions of \mathcal{L}_0 to logic programming. Details of the RCS domain and specifics of the translation are given. The translation is shown to be sound and complete for queries under Prolog and XSB. Computational examples are given which show the feasibility of this approach.

1 Introduction

In recent years a considerable amount of research has been done in the area of action languages [BGW], [Tur97], [BGP97]. Over the past year we have been working under a contract with United Space Alliance to use such languages to model subsystems of the Space Shuttle. The goal of the work under the contract is to provide tools to help plan for correct operation of the Shuttle in situations with multiple failures. In general there are pre-scripted plans for shuttle operation in cases of single failures, but not for cases of multiple failures. If such a case arises, ground controllers must develop plans to continue shuttle operation through the end of the mission.

For our initial research the Reaction Control System (RCS) of the Space Shuttle was chosen. A domain description for the RCS was created and tested. In this paper we discuss the action language \mathcal{L}_0 , how the RCS was modeled using this language, how the domain can be translated into a logic program, and results of computational examples under Prolog and XSB. Soundness and completeness results are also given.

In Section 2 the syntax and semantics of action language \mathcal{L}_0 will be discussed. Section 3 contains a general translation from domains in \mathcal{L}_0 to logic programs. In Section 4 we present details about the Reaction Control System of the Space Shuttle and how it has been modeled. Information about the performance of the program under Prolog and XSB is also discussed. The conclusions and topics for future work are in sections 5.

2 Syntax and Semantics of \mathcal{L}_0

Our description of dynamic domains will be based on the formalism of action languages. Such languages can be thought of as formal models of the parts of the natural language that are used for describing the behavior of dynamic domains. An action language can be represented as the sum of two distinct parts: an “action description language” and an “action query language”. A set of propositions in an action description language describes the effects of actions on states. Mathematically, it defines a transition system with nodes corresponding to possible states and arcs which are labeled by actions from the given domain. An arc σ_1, a, σ_2 indicates that an execution of action a in state σ_1 may result in the domain moving to the state σ_2 . An action query language serves for expressing properties of paths¹ within a given transition system. The syntax of such a language is defined by two classes of syntactic expressions: *axioms* and *queries*. The semantics of action description language is defined by specifying, for every transition diagram T of signature Σ , every set Γ of axioms, and every query Q , whether Q is a consequence of Γ in T . In this section we define a simple action language, \mathcal{L}_0 , which can be viewed as a sum of action description language \mathcal{A}_0 and query description language \mathcal{Q}_0 . We assume a fixed signature Σ_0 which consists of two disjoint, nonempty sets of symbols, a set, \mathbf{F} , of fluents and a set, \mathbf{A} , of actions. By fluent literals we mean fluents and their negations. The negation of $f \in \mathbf{F}$ will be denoted by $\neg f$. A set of fluent literals, S , is called *complete* if for any $f \in \mathbf{F}$, $f \in S$ or $\neg f \in S$. A set of fluent literals, S , is called *consistent* if there is no fluent f such that $f \in S$ and $\neg f \in S$. For a literal, l , the complement of l (denoted \bar{l}) is f if $l = \neg f$ and $\neg l$ otherwise.

2.1 Action Description Language \mathcal{A}_0

A *domain description* of \mathcal{A}_0 is an arbitrary set of propositions of the form:

1. *causes*($a, l_0, [l_1, \dots, l_n]$);
2. *causes*($[l_1, \dots, l_n], l_0$);
3. *impossible*($a, [l_1, \dots, l_n]$);

where l 's are fluent literals and a is an action. In each of the propositions above, l_0 is called the *head* of the proposition and $[l_1, \dots, l_n]$ is called the *body* of the proposition. A proposition of the type (1) says that, if the action a were to be executed in a situation in which l_1, \dots, l_n hold, the fluent literal l_0 will be caused to hold in the resulting situation. Such propositions are called *dynamic causal laws*. A proposition of the type (2), called a *static causal law*, says that the truth of fluent literals, l_1, \dots, l_n , in an arbitrary situation, s , is sufficient to cause the truth of l_0 in that situation. A proposition of the type (3) says that action a cannot be performed in any situation in which l_1, \dots, l_n hold.

¹ By a path of a transition system T we mean a sequence $\sigma_0, a_1, \sigma_1, \dots, a_n, \sigma_n$ such that for any $1 \leq i < n$, $\sigma_i, a_{i+1}, \sigma_{i+1}$ is an arc of T . σ_0 and σ_n are called initial and final states of the path respectively.

In addition to the propositions above, we allow *definition propositions* which will be viewed as a shorthand for specific sets of static causal laws. Definition propositions have the form:

$$definition(l_0, [l_1, \dots, l_n])$$

where l_0, \dots, l_n are fluent literals. The following restrictions apply to the use of such propositions:

1. Neither l_0 or \bar{l}_0 belong to the head of any of the static or dynamic causal laws.
2. For any fluent $f \in \mathbf{F}$, there may be definition propositions with either f or $\neg f$ in the head but not both.

For a fluent literal, l_0 , the meaning of a set, S , of such propositions, $\{definition(l_0, \beta_1), \dots, definition(l_0, \beta_n)\}$, can be read as “if there is a proposition, $definition(l_0, \beta_i)$, in S such that β_i holds in the current situation then l_0 also holds, otherwise \bar{l}_0 holds.”

As was mentioned, definition propositions are a shorthand for a larger set of static causal laws. If $\{definition(l_0, \beta_1), \dots, definition(l_0, \beta_n)\}$ is the set of all definition propositions whose head is l_0 then they can be replaced by the following static causal laws:

1. For each proposition, $definition(l, \beta_i)$, add a static causal law $causes(\beta, l)$.
2. For each set of literals, θ , such that:
 - (a) θ is consistent,
 - (b) for each β_i there exists a literal $l \in \beta_i$ such that $\bar{l} \in \theta$, and
 - (c) there is no subset of θ which satisfies conditions (a) and (b),
 add a static causal law $causes(\theta, \bar{l})$.

Example 1. The set of definition propositions:

$$definition(l, [p]).$$

$$definition(l, [q, \neg r]).$$

can be replaced by the following equivalent set of static causal laws:

$$causes([p], l).$$

$$causes([q, \neg r], l).$$

$$causes([\neg p, \neg q], \neg l).$$

$$causes([\neg p, r], \neg l).$$

In the remainder of the paper, when discussing domain descriptions in general, we will assume that definition propositions have been replaced by the corresponding static causal laws.

A domain description \mathcal{D} of \mathcal{A}_0 defines a transition diagram describing effects of actions on the possible states of the domain. By a *state* we mean a consistent set σ of fluent literals such that

1. σ is complete;
2. σ is *closed* under the static causal laws of \mathcal{D} , i.e. for any static causal law (2) of \mathcal{D} , if $\{l_1, \dots, l_n\} \subseteq \sigma$ then $l_0 \in \sigma$.

States serve as the nodes of the diagram. Nodes σ_1 and σ_2 are connected by a directed arc labeled by an action a if σ_2 may result from executing a in σ_1 . The set of all states that may result from doing a in a state σ will be denoted by $res(a, \sigma)$. Precisely defining this set for increasingly complex domain descriptions seems to be one of the main difficulties in the development of action theories. In case of domain descriptions from \mathcal{A}_0 we will use the approach suggested in [Tur97](#).

We will need the following auxiliary definitions. We say that an action, a , is *prohibited* in a state, σ , if \mathcal{D} contains a statement *impossible*($a, [l_1, \dots, l_n]$) such that $[l_1, \dots, l_n] \subseteq \sigma$.

Let F be a set of fluent literals of \mathcal{D} . By the *causal closure* of F we mean the least superset, $Cn_R(F)$, of F closed under the static causal laws of \mathcal{D} . By $E(a, \sigma)$ we denote the set of all fluent literals, l_0 , for which there is a dynamic causal law *causes*($a, l_0, [l_1, \dots, l_n]$) in \mathcal{D} such that $[l_1, \dots, l_n] \subseteq \sigma$.

Now we are ready for the main definition of this section. We say that a state σ' may result from doing action a in a state σ if

1. a is not prohibited in σ ;
2. σ' satisfies the condition $\sigma' = Cn_R((\sigma \cap \sigma') \cup E(a, \sigma))$

A domain description is called *deterministic* if for any action, a , and state, σ , there is at most one state, σ' , satisfying that above conditions. A domain description is called *qualification-free* if $res(a, \sigma) = \emptyset$ iff a is prohibited in σ .

2.2 Query Description Language \mathcal{Q}_0

The query language \mathcal{Q}_0 consists of two types of expressions: axioms and queries. Axioms of \mathcal{Q}_0 have a form

$$initially(l)$$

where l is a fluent literal. A collection of axioms describes the set of fluents known to be true in the initial situation. A set of axioms, Γ , is said to be *complete* if for any fluent $f \in \mathbf{F}$, either *initially*(f) $\in \Gamma$ or *initially*($\neg f$) $\in \Gamma$, but not both.

A query of \mathcal{Q}_0 is a statement of the form

$$holds(l, \alpha)$$

where l is a fluent literal and α is a sequence of actions. The statement says that α can be executed in the initial situation and, if it were, then fluent literal l would be true afterwards. To give the semantics of \mathcal{Q}_0 we need to interpret its statements with respect to a transition system, T , with signature Σ_0 . An *interpretation* is a partial function, Ψ , from sequences of actions from Σ_0 into paths of T which satisfy the following conditions:

1. the domain of Ψ is not empty;
2. if Ψ is defined on α , and β is a prefix of α , then $\Psi(\beta)$ is a prefix of $\Psi(\alpha)$.

Notice that the empty sequence, $[\]$, of action always belongs to the domain of Ψ .

We say that an axiom, *initially*(l), is true in an interpretation, Ψ , if $l \in \Psi([\])$; a set, Γ , of axioms is true in Ψ if every element of Γ is true in Ψ ; a query, *holds*(l, α), is true in Ψ if $\Psi(\alpha)$ is defined and its final state contains l . We say that a query, Q , is a consequence of a collection, Γ , of axioms with respect to a transition system, T , (denoted $\Gamma \models_T Q$) if Q is true in every interpretation in which Γ is true. We say that a query, Q , is a consequence of a collection, Γ , of axioms with respect to a domain description, \mathcal{D} , if $\Gamma \models_T Q$ for the transition system, T , defined by \mathcal{D} . Finally, we say that a collection of axioms, Γ , is *consistent* with respect to a domain description, \mathcal{D} , if there is at least one interpretation, Ψ , of the transition system of \mathcal{D} , in which Γ is true.

3 Translation to Logic Programming

Our methodology for computing the entailment relation of \mathcal{L}_0 will be to translate our domain description and axioms into a declarative logic program (under answer set semantics [GL91]). A domain description, \mathcal{D} , can simply be viewed as a logic program. To facilitate computation, we will add several facts to \mathcal{D} . First, we will divide the fluents of \mathbf{F} into two sets, \mathbf{F}_d and \mathbf{F}_f . \mathbf{F}_d contains those fluents for which either the fluent or its complement occurs in the head of a definition proposition. \mathbf{F}_f contains those which do not. For each fluent in \mathbf{F}_d we will add to \mathcal{D} one of the following two facts.

If f occurs in the head of a definition proposition then we add the fact

defined_literal(f).

otherwise we add

defined_literal($\neg f$).

For each fluent in \mathbf{F}_f we will add both

frame_literal(f).

and

frame_literal($\neg f$).

The set of axioms, Γ , is also viewed as a logic program. These two programs together, however, are not enough. We will also need a collection of domain independent axioms. The domain independent axioms will form a third logic program. We will denote this program as IID .

3.1 Domain Independent Axioms

In this section we will discuss the domain independent axioms, IID , in detail. In order to capture the semantics of \mathcal{L}_0 we must be able to correctly entail queries.

The first rules we present are not needed in the logic program per se, but will be needed when the logic program is later translated to run under Prolog or XSB. The rules simply state when two literals are contrary.

$$\begin{aligned} & \text{contrary}(\neg F, F). \\ & \text{contrary}(F, \neg F). \end{aligned}$$

In order to correctly ensure that a sequence of action is possible, we have the following rules which expands the impossibility propositions of our domain description to sequences of actions:

$$\begin{aligned} \text{impossible}([A \mid R]) &\leftarrow \text{impossible}(A, P), \\ &\quad \text{hold}(P, R). \\ \text{impossible}([A \mid R]) &\leftarrow \text{impossible}(R). \end{aligned}$$

The rules state that a sequence of actions is impossible if either the last action in the sequence is impossible or if the rest of the sequence is impossible.

The following rule captures the fact that if we have an axiom, *initially*(*l*), then *l* holds in the state in which no actions have been performed.

$$\text{holds}(L, [\]) \leftarrow \text{initially}(L).$$

Next we have a rule that states that “a literal, *L*, holds in the state that results from performing action, *A*, after a sequence of actions, *R*, if there is a dynamic causal law with action, *A*, and head, *L*, for which each literals in its body, *P*, held after performing *R*.” Note that the predicate *frame_literal*(*L*) is used to restrict the rule to literals which do not appear in the heads of defined predicates. Again, this is not required for the logic program but rather for improved computation under Prolog and XSB. We will see this technique used in several other rules below.

$$\begin{aligned} \text{holds}(L, [A \mid R]) &\leftarrow \text{frame_literal}(L), \\ &\quad \text{causes}(A, L, P), \\ &\quad \text{hold}(P, R), \\ &\quad \text{not impossible}([A \mid R]). \end{aligned}$$

For static causal laws we have the following rule which states that “a literal, *L*, holds in the state resulting from performing a sequence of actions, *S*, if there is a static causal law with *L* as the head and for which each of the literals in its body, *G*, also hold after performing *S*.”

$$\begin{aligned} \text{holds}(L, S) &\leftarrow \text{frame_literal}(L), \\ &\quad \text{causes}(G, L), \\ &\quad \text{hold}(G, S), \\ &\quad \text{not impossible}(S). \end{aligned}$$

The next pair of rules state when a set of literals hold.

$$\begin{aligned} \text{hold}([\], _). \\ \text{hold}([H \mid T], A) &\leftarrow \text{holds}(H, A), \\ &\quad \text{hold}(T, A). \end{aligned}$$

The first says that an empty set of literals hold (trivially) in any situation. The second states that a set of literals hold after a sequence of actions if each fluent in the set holds after that sequence of actions.

The next two rules are concerned with definition propositions. Together they state that “if there is a definition proposition with head L then, if the literals in its body, P , hold as a result of performing a sequence of actions, S , then L holds after performing S . If there is no definition proposition whose preconditions hold, then \bar{L} holds.”

$$\begin{aligned} \text{holds}(L, S) \leftarrow & \text{defined_literal}(L), \\ & \text{definition}(L, P), \\ & \text{hold}(P, S), \\ & \text{not impossible}(S). \end{aligned}$$

$$\begin{aligned} \text{holds}(F, S) \leftarrow & \text{contrary}(F, G), \\ & \text{defined_literal}(G), \\ & \text{not holds}(G, S), \\ & \text{not impossible}(S). \end{aligned}$$

The “common-sense law of inertia” [MH69] is captured by the following rule which states that fluents are normally not changed by performing actions. We use an “abnormality predicate” [McC86] to block the rule when an action does cause a change in the value of the fluent.

$$\begin{aligned} \text{holds}(L, [A \mid R]) \leftarrow & \text{frame_literal}(L), \\ & \text{holds}(L, R), \\ & \text{not ab}(L, A, R), \\ & \text{not impossible}([A \mid R]). \end{aligned}$$

Note that the inertia rule does not apply to fluents that occur in the heads of definition propositions. Such “defined” fluents are always determined by the propositions themselves and therefore inertia never applies.

Finally, the last two rules state that “a literal, L , is abnormal with respect to the inertia axiom if \bar{L} was caused by either a dynamic or static causal law as a result of performing action, A , in the state that resulted from performing action sequence, R .”

$$\begin{aligned} \text{ab}(F, A, R) \leftarrow & \text{contrary}(F, G), \\ & \text{causes}(A, G, P), \\ & \text{hold}(P, R). \end{aligned}$$

$$\begin{aligned} \text{ab}(F, A, R) \leftarrow & \text{contrary}(F, G), \\ & \text{causes}(P, G), \\ & \text{hold}(P, [A \mid R]). \end{aligned}$$

3.2 Correctness of Domain Independent Axioms

We now present a theorem stating that the above axioms are correct.

Theorem 1. For any deterministic, qualification free, domain description \mathcal{D} , transition system T defined by \mathcal{D} , complete, consistent, set of axioms Γ , and query, $holds(l, \alpha)$, $\Gamma \models_T holds(l, \alpha)$ iff $\mathcal{D} \cup \Gamma \cup \Pi\mathcal{D} \models holds(l, \alpha)$

4 The RCS Domain

The job of the Reaction Control System (known as the RCS) is primarily to provide maneuvering capabilities for the Space Shuttle while it is in orbit. It is composed of three subsystems, the Forward RCS, Left RCS, and Right RCS. Each subsystem consists of a fuel tank, an oxidizer tank, two pressurized helium tanks, a number of maneuvering jets, a series of pipes connecting the components, valves to control flow between the components, power and logic circuits, drivers, multiplexer/de-multiplexers, and switches which control the valves and circuits. The Right and Left RCS's also have cross-feeds which allow fuel and oxidizer from one to be used by the other in case of a major failure in that subsystem. In order to ready one of the jets on the RCS to fire, the power, logic, and drivers for the jet must be operational and there must be an open pipeline to both a pressurized source of fuel and a pressurized source of oxidizer. Several jets, each from specific subsets of the jets, must be ready to fire in order to be ready to perform a maneuver.

4.1 The Propositions of the RCS Domain

Rather than writing all the propositions of the domain description by hand, we will generate the domain description by using a logic program. This is necessary due to the enormous size of the domain. To illustrate this, consider the definition propositions which determine if a jet of the RCS is ready to fire in a given situation. The RCS contains 44 jets, many of which can be readied in several different ways. As a result, one would need several hundred such propositions in the domain description. In the program presented below, all of the propositions concerning readying a jet to fire are generated by one rule. The savings does not come without a cost. A large number of facts are required in conjunction with the rules to correctly model the domain. There are over 200 facts for the predicate 'link' which describes how the pipes in the fuel systems are connected. There are also many facts which define which objects are of what type. These are used in conjunction with other rules to generate the lists of "frame fluents" and "defined fluents." In addition to the rules described below, the program contains approximately 700 of these simple facts and over 20 auxiliary rules. Due to space considerations, these facts and rules will not be presented here. We will, however, present the rules which are ultimately used to generate the propositions of the domain description. First we will present the rules for creating the dynamic causal laws of this domain.

There is only one type of action in this domain, changing the position of a switch. Performing an action of flipping a switch to a position “causes” the switch to be in the new position. These dynamic causal laws are captured by the following rule, with actual causal laws being ground instances of the rule where S is a switch and P is a valid position.

$$\text{causes}(\text{flip}(S, P), \text{position}(S, P), []).$$

Flipping a switch to a given position also ensures that the switch is in no other position after performing the action. The rule below shows this effect.

$$\text{causes}(\text{flip}(S, P1), \neg \text{position}(S, P2), []) \leftarrow \text{diff}(P1, P2).$$

Next we have a rule stating that it is impossible to flip a switch to a position it is already in. Note that, since there are 50 switches in the RCS subsystem, this rule cuts the number of executable actions in a situation from 100 down to 50.

$$\text{impossible}(\text{flip}(S, P), [\text{position}(S, P)]).$$

Now we present the rules for generating the static causal laws of this domain. Many of the switches in the RCS are used to control the position of valves. It may seem strange that the dynamic causal laws above did not capture this. The reason for this seemingly odd situation can be explained by a simple example.

Imagine we wish to model the operation of an ordinary lamp. One is tempted to have a dynamic causal law stating that if the switch is turned on, then the light comes on. But what if the bulb is burned out? We could add a precondition to the law stating that it only applies when the bulb is good. This, however, is only half the battle if we have an action to change the bulb. We would then need a dynamic causal law stating that changing the bulb causes the light to be on if the switch is on. Suppose we then update the domain by saying that the lamp can be unplugged and we add a new action to plug in the lamp. Both of the previous dynamic causal laws need to be updated and a new law needs to be added.

Now consider a different approach. The dynamic causal laws simply state that turning the switch on causes the switch to be on and changing the bulb causes the bulb to be good. We then add a static causal law stating that if the switch is on and the bulb is good then the light is on. Now, in order to add the information about plugging in the lamp, we simply add a new dynamic causal law stating that plugging in the lamp causes it to be plugged in. We also must modify the one existing static causal law to reflect that the lamp must be plugged in for the light to be on. This approach is preferable for two primary reasons. First, as was shown by the example, it is more elaboration tolerant. The second reason deals with the initial situation. Using the first approach, we could have a consistent set of axioms which stated that the light was initially on and the bulb was initially burned out. Using the second approach, this initial situation is not consistent since it is not closed with respect to the static causal laws of the domain.

We see the second approach used when combining the dynamic causal laws shown previously with the following two rules:

$$\begin{aligned} &causes([position(S, open), \\ &\quad \neg non_functional(open, S), \\ &\quad \neg stuck(closed, V)], \\ &\quad open(V)) \leftarrow switch_controls(S, V). \end{aligned}$$

$$\begin{aligned} &causes([position(S, closed), \\ &\quad \neg non_functional(closed, S), \\ &\quad \neg stuck(open, V)], \\ &\quad \neg open(V)) \leftarrow switch_controls(S, V). \end{aligned}$$

The first of the two rules above states that, if a valve is functional, it is not stuck closed, and the switch controlling it is in the open position, then the valve will be open. The second is a similar rule stating the conditions under which a valve is closed.

In order to provide fuel to a jet, a propulsion tank must be properly pressurized. The propulsion tanks are pressurized by opening valves between the propulsion tank and a pressurized helium tank. The static laws concerning this are described by the rule below. The rule states that if a helium tank has correct pressure, there is an open path to another tank (which will be a propulsion tank), and there are no paths to a leak, then the second tank also has correct pressure.

$$\begin{aligned} &causes([correct_pressure(S), \\ &\quad open_path(S, T), \\ &\quad \neg path_to_leak(S)], \\ &\quad correct_pressure(T)) \leftarrow source_of(S, helium), \\ &\quad fuel_tank(T). \end{aligned}$$

Next we look at definition propositions. Recall that such propositions are merely a shorthand for a larger set of causal laws.

The first set of definition propositions describe what it means to have an open path in the RCS. Intuitively, we say that there is an open path between two points if a path exists between the points and all valves on the interior of the path are open.

$$definition(open_path(X, Y), [all_open(Z)]) \leftarrow valve_path(X, Y, Z).$$

For the above definition we need to define when a set of valves are all open. This is done by the following two definitions.

$$\begin{aligned} &definition(all_open([], [])). \\ &definition(all_open([H \mid T]), [open(H), all_open(T)]) \leftarrow valve(H). \end{aligned}$$

There is a path from a point to a leak in the system if there is an open path from that point to a valve that is open and leaking.

definition(*path_to_leak*(*S*),
 [*leaking*(*X*), *open*(*X*), *open_path*(*S*, *X*)] \leftarrow *valve*(*X*)).

The RCS contains many power, control, and logic buses. For drivers and logic circuits to work, they need the appropriate buses to be working. For safety reasons, some of the drivers and logic circuits can use either of a given pair of such buses to be operational. In order to model this, we define what it means for at least one of a pair of buses to be working.

definition(*at_least_1*(*A*, *B*), [\neg *bad*(*A*)]).
definition(*at_least_1*(*A*, *B*), [\neg *bad*(*B*)]).

In order for the drivers and logic circuits to work, the corresponding power switch must be on and operational. This is captured by the proposition:

definition(*powered*(*X*),
 [*position*(*X*, *on*), \neg *non_functional*(*on*, *X*)]).

The follow definition proposition states when a particular driver, driver *rjdf2a*, is “ok”. There are nine drivers in the RCS, each with its own proposition. To save space, only one will be presented here. The proposition states that if logic circuit *f3* is ok, control bus *ca2* is working, at least one power bus in each of the proper pairs is operational, and the driver is turned on, then driver *rjdf2a* is operational, otherwise, it is not.

definition(*ok*(*rjdf2a*),
 [*ok*(*f3*), \neg *bad*(*ca2*),
 at_least_1(*flc1*, *flc3*),
 at_least_1(*fpc1*, *fpc2*),
 powered(*f3dr*)]).

We have similar propositions for each of the logic circuits. Again, only one of the nine such propositions is presented here. The logic circuit, *f3*, is operational if and only if control bus *ca1* is operational, power bus *fpc1* and/or power bus *fpc3* is working, and the logic circuit is powered on.

definition(*ok*(*f3*), [\neg *bad*(*ca1*), *at_least_1*(*fpc1*, *fpc3*), *powered*(*f3lo*)]).

We now get to the primary propositions in our domain, the definition of when a single jet is ready to fire, and the definitions for when the shuttle is ready for a given maneuver. A jet of the RCS is ready to fire if it is not damaged, it has open, non-leaking paths to both a pressurized source of fuel and a pressurized source of oxidizer, its associated driver is operational, and its multiplexer/de-multiplexer is working.

definition(*ready_to_fire*(*J*), [\neg *damaged*(*J*),
 correct_pressure(*S1*),
 open_path(*J*, *S1*),
 \neg *path_to_leak*(*S1*),

$$\begin{aligned}
& \text{correct_pressure}(S2), \\
& \text{open_path}(J, S2), \\
& \neg \text{path_to_leak}(S2), \\
& \text{ok}(D), \\
& \neg \text{bad}(W)) \leftarrow \text{source_of}(S1, \text{fuel}), \\
& \qquad \text{source_of}(S2, \text{oxidizer}), \\
& \qquad \text{assoc_drv}(J, D), \\
& \qquad \text{assoc_mdm}(J, W).
\end{aligned}$$

By firing the jets of the RCS, the Space Shuttle is able to adjust its speed and rotation. While very complex maneuvers are possible, they can all be broken down into a small set of basic maneuvers. The basic maneuvers can be separated into two categories, translation maneuvers and rotation maneuvers. Translation maneuvers affect the speed of the shuttle along its X, Y, or Z axis. Rotation maneuvers affect the speed at which the shuttle rotates about one of the three axes. There is one basic maneuver of each type for each direction along each axis, therefore there are twelve such basic maneuvers. We will only present one of the maneuver propositions here.

The *+ roll* maneuver allows the shuttle to change the speed of rotation along the X axis. The proposition describing such maneuvers is rather straight forward. It states that to be ready to perform this maneuver there must be two jets ready to fire; one on the Left RCS which is pointing downward, and one on the Right RCS which is pointing upward.

$$\begin{aligned}
& \text{definition}(\text{ready_for_maneuver}(\text{plus_roll}), \\
& \quad [\text{ready_to_fire}(X), \text{ready_to_fire}(Y)]) \leftarrow \text{jet_of}(X, \text{left_rcs}), \\
& \qquad \text{direction}(X, \text{down}), \\
& \qquad \text{jet_of}(Y, \text{right_rcs}), \\
& \qquad \text{direction}(Y, \text{up}).
\end{aligned}$$

As was mentioned, the program described above will be used to generate the RCS domain description. Recall that a domain description is a set of propositions of \mathcal{A}_0 (including definition propositions). The RCS domain description, \mathcal{D} , contains all propositions, p , where p is a consequence of the program above.

4.2 Initial Situations

In order to use this domain description, we must provide the initial situation. Recall that a collection of axioms, Γ , of the query description language, describe the set of fluents known to be true in the initial situation. For use with the RCS domain, we wish to consider only complete collections of axioms, therefore, we must have an axiom for each fluent or its negation. In actual use, we may be able to extract the information needed to generate the initial situation from the Space Shuttle's telemetry data. At this time, and in cases where we wish to explore hypothetical situations, we generate the initial situation using a program.

While this is the same approach we used to generate the domain description, the programs used to generate initial conditions are much smaller. This is due

in part to the fact that most fluents, such as those used to denote malfunctions in the shuttle, usually maintain a certain truth value. If the ground instances of a fluent, $f(\mathbf{X})$, are generally true in the initial situation, we can insert the following rule:

$$initially(f(\mathbf{X})) \leftarrow fluent(f(\mathbf{X})), not\ initially(\neg f(\mathbf{X}))$$

If the ground instances are usually false we add the rule:

$$initially(\neg f(\mathbf{X})) \leftarrow fluent(f(\mathbf{X})), not\ initially(f(\mathbf{X}))$$

We then only need to explicitly list those instances of the fluent that are contrary to the norm. The predicate ‘fluent’ was added to the rules so that the program would not flounder when run under Prolog.

Such rules can be used for fluents like the ‘open(valve)’ fluent in the RCS domain. If the majority of the valves are open in the initial situation, the first rule can be used and only the closed valves need to be listed.

Using this approach, a program to generate an initial situation would be a collection of rules as above, together with a collection of axioms which are given explicitly. Given such a program, an axiom, a , will belong to our set of axioms, Γ , iff a is a consequence of the program.

4.3 Soundness and Completeness

In this section results concerning the soundness and completeness of the RCS domain with respect to logic programming and execution under Prolog and XSB are presented. First we present a theorem concerning properties of the RCS domain.

Theorem 2. The RCS domain, \mathcal{D} , is qualification free and deterministic.

This leads to the following corollary:

Corollary 1. Let \mathcal{D} be the RCS domain description, T be the transition system defined by \mathcal{D} , and Γ be any complete, consistent, set of axioms, then for any query, $holds(l, \alpha)$, $\Gamma \models_T holds(l, \alpha)$ iff $\mathcal{D} \cup \Gamma \cup \Pi \mathcal{D} \models holds(l, \alpha)$.

Proof: Follows directly from theorems 1 and 2.

Before we present a theorem for correctness under Prolog and XSB, we first need to present the following definitions. For a fluent literal, l , we will define l^+ as l if $l = f$ for some fluent f and $neg(l)$ if $l = \neg f$ for some fluent f . For a query, $q = holds(l, \alpha)$, $q^+ = holds(l^+, \alpha)$. For a logic program Π , Π^+ is the logic program that results from replacing each literal, l in Π by l^+ .

Theorem 3. Let q be a ground query and $\Pi = \mathcal{D} \cup \Gamma \cup \Pi \mathcal{D}$ be a logic program, where \mathcal{D} is the RCS domain description and Γ is a complete, consistent set of axioms, then given the program, Π^+ , and query, q^+ , Prolog (and XSB) answers yes iff $\Pi \models q$.

4.4 Usage Results

We have given results about the soundness and completeness of our implementation. We will now present some details about the performance. The program was run under both Prolog and XSB and on both PCs and Sun workstations. For Prolog, several modifications were made to the code. “Cuts” were added to the rules for *holds* and the computation of *impossible* was separated from computation of *holds*. These modifications do not effect the soundness and completeness of the translation. The intended use of the current system is to verify if a given plan readies the shuttle for a given maneuver. Tests using a variety of sample plans and maneuvers were executed. Generally, given a fairly normal initial situation with few malfunctioning components, it requires a plan consisting of approximately 20 actions to ready the shuttle for a maneuver. In the best case, Prolog is slightly faster than XSB. The worst performance occurs when the given plan does not achieve the goal. Under Prolog the time required to verify that such a plan did not achieve the goal was as high as 4 1/2 minutes. XSB, with its ability to ‘table’ predicates, did much better. Its worst case performance was around 6 seconds. It should also be noted that, due to tabling, if subsequent queries are asked without exiting XSB, a tremendous performance improvement is seen, with similar queries returning with a yes or no answer almost instantly.

5 Conclusions

The system presented in this paper could provide flight controllers with an automated system to verify plans for operation of the RCS, a capability which they do not currently possess. Due to the small size of the code and the portability of Prolog and XSB, such tools could also be taken on the shuttle itself. This would provide the astronauts the ability to verify plans in case they were cut off from the system experts on the ground. The system is provenly correct and relatively fast.

The domain descriptions can also be easily read and verified, not only by persons with expertise in logic programming, but also by experts in the domain being modeled. Our contact at United Space Alliance, a former flight controller for the RCS system, was able to spot several minor errors in an early draft simply by reading the source code. In addition, he felt the action language was so intuitive and easy to use that he is currently writing a domain description for the Shuttle’s OMS subsystem using the RCS domain as a guide.

In order to provide an even more useful tool, we are currently working to expand this system in two directions. Our first goal is to expand the system by adding a diagnostic component. This would be used when a sequence of actions was actually performed but unexpected results were observed. Our goal is to have a system that could be used to determine what failure or failures most likely occurred which would explain the observed behavior.

The second direction for further work concerns planning. While the current system is a good plan checker, it would be beneficial to be able to generate plans

as well. We would like to be able to provide the system with the current situation and a goal and have it generate a plan to achieve the goal from that situation.

While we expand the current system, we wish to ensure that such expansions provide a general framework for use with other domains. We would like to create a domain independent system with the features mentioned above. Ultimately, we wish to create an interface which would allow an expert in a given domain to create a domain description with only minor knowledge of logic programming.

Acknowledgments

The author would like to thank Michael Gelfond and Matt Barry for their contributions, discussions, and comments. This work was partially supported by a contract with United Space Alliance.

References

- [BGP97] Chitta Baral, Michael Gelfond, and Alessandro Provetti. Representing Actions: Laws, Observations, and Hypothesis. In *Journal of Logic Programming*, Vol. 31, No. 1-3, pp. 245-298, 1997.
- [BGW] Chitta Baral, Michael Gelfond, and Richard Watson. Reasoning About Actual and Hypothetical Occurrences of Concurrent and Non-deterministic Actions, in *Theoretical Approaches to Dynamic Worlds*, edited by Bertram Fronhofer and Remo Pareschi (to appear).
- [GL91] Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. In *New Generation Computing*, 9(3/4), pp. 365–387, 1991.
- [GL92] Michael Gelfond and Vladimir Lifschitz. Representing Actions in Extended Logic Programs. In *Proc. of Joint International Conference and Symposium on Logic Programming*, pp. 559-573, 1992.
- [McC86] John McCarthy. Applications of Circumscription to Formalizing Common Sense Knowledge. In *Artificial Intelligence*, 26(3), pp. 89–116, 1986.
- [MH69] John McCarthy and Patrick Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pp. 463–502. Edinburgh University Press, Edinburgh, 1969.
- [Tur97] Hudson Turner. Representing actions in logic programs and default theories: A situation calculus approach. In *Journal of Logic Programming*, Vol. 31, No. 1-3, pp. 245-298, 1997.

Developing a Declarative Rule Language for Applications in Product Configuration

Timo Soininen¹ and Ilkka Niemelä²

¹ Helsinki University of Technology, TAI Research Center and Lab. of Information Processing Science, P.O.Box 9555, FIN-02015 HUT, Finland

`Timo.Soininen@hut.fi`

² Helsinki University of Technology, Dept. of Computer Science and Eng., Laboratory for Theoretical Computer Science, P.O.Box 5400, FIN-02015 HUT, Finland

`Ilkka.Niemela@hut.fi`

Abstract. A rule-based language is proposed for product configuration applications. It is equipped with a declarative semantics providing formal definitions for main concepts in product configuration, including configuration models, requirements and valid configurations. The semantics uses Horn clause derivability to guarantee that each element in a configuration has a justification. This leads to favorable computational properties. For example, the validity of a configuration can be decided in linear time and other computational tasks remain in **NP**. It is shown that CSP and dynamic CSP can be embedded in the proposed language which seems to be more suitable for representing configuration knowledge. The rule language is closely related to normal logic programs with the stable model semantics. This connection is exploited in the first implementation which is based on a translator from rules to normal programs and on an existing high performance implementation of the stable model semantics, the Smodels system.

1 Introduction

Product configuration has been a fruitful topic of research in artificial intelligence for the past two decades (see, e.g. [10,15,18]). In the last five years product configuration has also become a commercially successful application of artificial intelligence techniques. Knowledge-based systems (KBS) employing techniques such as constraint satisfaction (CSP) [19] have been applied to product configuration. However, the product configuration problem exhibits dynamic aspects which are difficult to capture in, e.g., the CSP formalism. The choices form chains where previous choices affect the set of further choices that need to be made. In addition, making a choice needs to be justified by a chain of previous choices. This has led to the development of extensions of the CSP formalism, such as dynamic constraint satisfaction (DCSP) [11] and generative constraint satisfaction (GCSP) [7].

In this paper, which is a revised version of [17], we present work-in-progress on developing a logic programming like rule language for product configuration

applications. The rule language is defined with the goal that relevant knowledge in the configuration domain can be represented compactly and conveniently. We provide a simple declarative semantics for the language which guarantees a justification for each choice.

We study the complexity of the relevant computational tasks for this language. The main result is that the task of finding a configuration is **NP**-complete and that the validity of a configuration can be checked in linear time. We also show that our language can be seen as a generalization of the CSP and DCSP formalisms. There are local and linear solution preserving mappings from the CSP and DCSP formalisms to the language, but mapping in the other direction is difficult. This is due to the difficulty of capturing justifications in CSP and to more expressive rules that seem difficult to capture in DCSP.

The semantics of the rule language is closely related to the declarative semantics of logic programs. This relation is exploited in developing the first implementation of the language. We present a solution preserving local and polynomial translation from the rule language to normal logic programs with the stable model semantics [6]. Our implementation is based on an existing high performance implementation of the stable model semantics for normal logic programs, the Smodels system [12,13]. For the implementation it is enough to build a front-end to the Smodels system realizing the translation to normal programs. In order to estimate the feasibility of our approach we study two simple configuration problems. We observe that such examples are straightforward to model in our language and that our implementation exhibits reasonable performance.

2 Product Configuration Domain

Product configuration is roughly defined as the problem of producing a specification of a product individual as a collection of predefined components. The inputs of the problem are a *configuration model*, which describes the components that can be included in the configuration and the rules on how they can be combined to form a working product, and *requirements* that specify some properties that the product individual should have. The output is a *configuration*, an accurate enough description of a product individual to be manufactured. The configuration must *satisfy* the requirements and be *valid* in the sense that it does not break any of the rules in the configuration model and it consists only of the components that have justifications in terms of the configuration model.

This definition of product configuration does not adequately capture all aspects of configuration problems. Missing features include representing and reasoning about attributes, structure and connections of components, resource production and use by components [15,17] and optimality of a configuration. Our definition is a simplification that nonetheless contains the core aspects of configuration problem solving. It is intended as the foundation on which further aspects of product configuration can be defined. Correspondingly, we use the term *element* to mean any relevant piece of information on a configuration. An element can be a component or information on, e.g., the structure of a product.

A *product configurator* is a KBS that is capable of representing the knowledge included in configuration models, requirements and configurations. In addition, it is capable of (i) *checking* whether a configuration is valid with respect to the configuration model and satisfies a set of requirements and/or (ii) *generating* one or all valid configuration(s) for a configuration model and a set of requirements.

Example 1. As an example of a configurable product, consider a PC. The components in a typical configuration model of a PC include different types of display units, hard disks, CD ROM drives, floppy drives, extension cards and so on. These have rules on how they can be combined with each other to form a working product. For example, a PC would typically be defined to have a mass storage which must be chosen from a set of alternatives, e.g. an IDE hard disk, SCSI hard disk and a floppy drive. A computer would also need a keyboard, which could have either a Finnish or United Kingdoms layout. Having a SCSI hard disk in the configuration of a PC would typically require that an additional SCSI controller is included in the configuration as well. In addition, a PC may optionally have a CD ROM drive. A configuration model for a PC might also define that unless otherwise specified, an IDE hard disk will be the default choice for mass storage.

The fundamental form of knowledge in a configuration model is that of a *choice* [18]. There are basically two types of choices. Either at least one or exactly one of alternative elements must be chosen. Whether a choice must be made may depend on some set of elements. Other forms of configuration knowledge include the following:

- A set of elements in the configuration *requires* some set of elements to be in the configuration as well [18].
- A set of elements are *incompatible* with each other [18].
- An element is *optional*. Optional elements can be chosen into a configuration or they can be left out.
- An element is a *default*. It is in the configuration unless otherwise specified.

3 Configuration Rule Language

In this section we define a configuration rule language **CRL** for representing configuration knowledge. The idea is to focus on interactions of the elements and not on details of a particular configuration knowledge modeling language. For simplicity, we have kept the number of primitives in the language low by focusing on choices and requires and incompatibility interactions. Extending the language with optional and default choices is straightforward (see Example 4).

The basic construction blocks of the language are propositional atoms, which are combined through a set of connectives into rules. We assume for simplicity that atoms can be used to represent elements adequately. We define a *configuration model* and *requirements* as sets of **CRL** rules. A *configuration* is defined as a set of atoms.

The syntax of **CRL** is defined as follows. The alphabet of **CRL** consists of the connectives “,”, “ \leftarrow ”, “|”, “ \oplus ”, “not”, parentheses and atomic propositions. The connectives are read as “and”, “requires”, “or”, “exclusive or” and “not”, respectively. The rules in **CRL** are of the form

$$a_1\theta \cdots \theta a_l \leftarrow b_1, \dots, b_m, \text{not}(c_1), \dots, \text{not}(c_n)$$

where $\theta \in \{ |, \oplus \}$, $a_1, \dots, a_l, b_1, \dots, b_m, c_1, \dots, c_n$ are atoms and $l \geq 0, m \geq 0, n \geq 0$. We refer to the subset of a set of rules R with exactly one atom in the head as *requires-rules*, R_r , rules with more than one atom in the head separated by “|” as *choice-rules*, rules with more than one atom in the head separated by “ \oplus ” as *exclusive choice-rules* R_e , and rules with no atoms in the head as *incompatibility-rules*, R_i . In the definitions below we treat requires-rules as a special case of choice-rules with only one alternative in the head.

Example 2. A very simple configuration model R_{PC} of the PC in Example □ (without the optional CD-ROM and default mass storage) could consist of the following rules:

$$\begin{aligned} & \text{computer} \leftarrow \\ & IDEdisk \mid SCSIdisk \mid floppydrive \leftarrow \text{computer} \\ & FinnishlayoutKB \oplus UKlayoutKB \leftarrow \text{computer} \\ & SCSIcontroller \leftarrow SCSIdisk \end{aligned}$$

Next we define when a configuration satisfies a set of rules and is valid with respect to a set of rules. We say that a configuration *satisfies requirements* if it satisfies the corresponding set of rules.

Definition 1. A configuration C satisfies a set of rules R in **CRL**, denoted by $C \models R$, iff the following conditions hold:

- (i) If $a_1 \mid \cdots \mid a_l \leftarrow b_1, \dots, b_m, \text{not}(c_1), \dots, \text{not}(c_n) \in R_r \cup R_e$, $\{b_1, \dots, b_m\} \subseteq C$, and $\{c_1, \dots, c_n\} \cap C = \emptyset$, then $\{a_1, \dots, a_l\} \cap C \neq \emptyset$.
- (ii) If $a_1 \oplus \cdots \oplus a_l \leftarrow b_1, \dots, b_m, \text{not}(c_1), \dots, \text{not}(c_n) \in R_e$, $\{b_1, \dots, b_m\} \subseteq C$, and $\{c_1, \dots, c_n\} \cap C = \emptyset$, then for exactly one $a \in \{a_1, \dots, a_l\}$, $a \in C$.
- (iii) If $\leftarrow b_1, \dots, b_m, \text{not}(c_1), \dots, \text{not}(c_n) \in R_i$, then it is not the case that $\{b_1, \dots, b_m\} \subseteq C$ and $\{c_1, \dots, c_n\} \cap C = \emptyset$ hold.

In order to define the validity of a configuration, we employ an operator R^C that is a transformation of a set of rules R in **CRL**.

Definition 2. Given a configuration C and a set of rules R in **CRL**, we denote by R^C the set of rules

$$\{a_i \leftarrow b_1, \dots, b_m : a_1\theta \cdots \theta a_l \leftarrow b_1, \dots, b_m, \text{not}(c_1), \dots, \text{not}(c_n) \in R, \theta \in \{ |, \oplus \}, a_i \in C, 1 \leq i \leq l, \{c_1, \dots, c_n\} \cap C = \emptyset\}$$

The result of the transformation is a set of Horn clauses if we interpret the symbols “ \leftarrow ” and “,” as classical implication and conjunction, respectively. Under this interpretation the reduct R^C has a unique least model, which we denote by

$\text{MM}(R^C)$. Notice that the least model of a set of Horn clauses coincides with the set of atoms logically entailed by them and also with the set of atoms derivable by interpreting them as inference rules. The intuition behind the transformation is that, given a choice-rule, if any of the alternatives in the head of the rule are chosen, then the reduct of the transformation includes a rule that can justify the choice (if the body of the rule can be justified). If some alternative is not chosen, then there is no need for the choice to be justified and consequently no corresponding rules are included. The default negation “not(\cdot)” is handled using a technique similar to that in the stable model semantics of logic programs [6].

Definition 3. *Given a configuration C and a set of rules R in **CRL**, C is R -valid iff $C = \text{MM}(R^C)$ and $C \models R$.*

The idea of the definition is as follows: the first fix-point condition guarantees that a configuration must be justified by the rules. All the things in the configuration are derivable from (the reduct of) the configuration rules. On the other hand, everything that can be derived using (the reduct of) the rules must be in the configuration. The second condition ensures that all the necessary choices have been made and all the requires and incompatibility-rules are respected.

Example 3. Consider the configuration model R_{PC} in Example 2, the simple set of requirements $\{FinnishlayoutKB \leftarrow\}$ and the configurations

$$\begin{aligned} C_1 &= \{computer, SCSIdisk, UKlayoutKB\} \\ C_2 &= \{computer, IDEdisk, FinnishlayoutKB, SCSIcontroller\} \\ C_3 &= \{computer, SCSIdisk, FinnishlayoutKB, SCSIcontroller\} \end{aligned}$$

The configuration C_1 does not satisfy the configuration model nor the requirements according to Definition 1 and thus it is not R_{PC} -valid, either. The configuration C_2 does satisfy the configuration model and the requirements. However, it is not R_{PC} -valid because the reduct $R_{PC}^{C_2}$ is

$$\{computer \leftarrow; IDEdisk \leftarrow computer; FinnishlayoutKB \leftarrow computer; SCSIcontroller \leftarrow SCSIdisk\}$$

The minimal model $\text{MM}(R_{PC}^{C_2}) = \{computer, IDEdisk, FinnishlayoutKB\}$ does not contain $SCSIcontroller$ and thus it is not equal to C_2 . The configuration C_3 is R_{PC} -valid and satisfies the requirements.

Example 4. Consider the following sets of rules:

$R_1 :$ $a \mid b \leftarrow c$ $c \leftarrow$	$R_2 :$ $a \mid b \leftarrow c$ $c \oplus c' \leftarrow d$ $d \leftarrow$	$R_3 :$ $a \mid b \leftarrow c$ $c \oplus c' \leftarrow d$ $a \leftarrow \text{not}(b), d$ $d \leftarrow$
--	--	---

The valid configurations with respect to R_1 are $\{c, a\}$, $\{c, b\}$ and $\{c, a, b\}$. The reducts of R_1 with respect to these configurations are $\{a \leftarrow c; c \leftarrow\}$, $\{b \leftarrow c; c \leftarrow\}$

and $\{a \leftarrow c; b \leftarrow c; c \leftarrow\}$, respectively. Clearly, the minimal models of these reducts coincide with the configurations and the configurations satisfy the rules in R_1 . On the other hand, if the latter rule is omitted, the only valid configuration is the empty configuration $\{\}$, since a and b cannot have a justification.

Although **CRL** does not include primitives for some typical forms of configuration knowledge such as *optional choices* and *default* alternatives, they can be captured fairly straightforwardly. The first two rules in R_2 demonstrate how to represent an optional choice-rule whose head consists of the atoms a and b and whose body is d . The valid configurations with respect to R_2 are $\{c', d\}$, $\{a, c, d\}$, $\{b, c, d\}$ and $\{a, b, c, d\}$. In this example either c or c' must be in a configuration. These additional atoms represent the cases where the choice is made and not made, respectively. Now, consider the rule set R_3 obtained by adding the rule $a \leftarrow \text{not}(b), d$ to R_2 . The valid configurations are now $\{c', a, d\}$, $\{c, a, d\}$, $\{c, b, d\}$ and $\{c, a, b, d\}$. This rule set represents a default choice (a is the default) which is made unless one of the alternatives is explicitly chosen.

4 Relationship to Logic Programming Semantics

The configuration rule language **CRL** resembles disjunctive logic programs and deductive databases. The main syntactic difference is that two disjunctive operators are provided whereas in disjunctive logic programming typically only one is offered. The semantics is also similar to logic programming semantics. The main difference is that leading disjunctive semantics (see, e.g., [3,5]) have minimality of models as a built-in property whereas our semantics does not imply subset minimality of configurations. The rule set R_1 above is an example of this. However, there are semantics allowing non-minimal models and, in fact, if we consider the subclass with one disjunctive operator, i.e. ordinary choice-rules, our notion of a valid configuration coincides with possible models introduced by Sakama and Inoue [14] for disjunctive programs. They observed that possible models of disjunctive programs can be captured with stable models of normal programs by a suitable translation of disjunctive programs to non-disjunctive programs [14]. Here we extend this idea to exclusive choice-rules and present a slightly different, more compact and computationally oriented translation.

Given a set of rules R in **CRL** the corresponding normal logic program is constructed as follows. The requires-rules R_r are taken as such. The incompatibility-rules R_i are mapped to logic program rules with the same body but a head f and a new rule $f' \leftarrow \text{not}(f')$, f is included where f, f' are new atoms not appearing in R . For each choice-rule

$$a_1 \mid \cdots \mid a_l \leftarrow b_1, \dots, b_m, \text{not}(c_1), \dots, \text{not}(c_n)$$

in R_c we include a rule $f \leftarrow b_1, \dots, b_m, \text{not}(c_1), \dots, \text{not}(c_n), \hat{a}_1, \dots, \hat{a}_l$ and for all $i = 1, \dots, l$, two rules

$$a_i \leftarrow \text{not}(\hat{a}_i), b_1, \dots, b_m, \text{not}(c_1), \dots, \text{not}(c_n) \quad \text{and} \quad \hat{a}_i \leftarrow \text{not}(a_i)$$

where $\hat{a}_1, \dots, \hat{a}_l$ are new atoms. Each exclusive choice-rule is translated the same way as an ordinary choice-rule except that we include additionally the set of rules of the form $f \leftarrow b_1, \dots, b_m, \text{not}(c_1), \dots, \text{not}(c_n), a', a''$ where $a' = a_i, a'' = a_j$ for some $i, j, 1 \leq i < j \leq l$. Note that the number of the additional rules is quadratic in the number of head atoms, but for ordinary choice-rules the translation is linear. Now the stable models of the program provide the valid configurations for the rules. The close correspondence implies that an implementation of the stable model semantics can be used for configuration tasks.

5 Complexity Issues

In this section we briefly consider the complexity of the following key decision problems in configuration: (i) **C-SAT**: decide whether a configuration satisfies a set of rules, (ii) **EXISTS**: determine whether there is a valid configuration for a set of rules, and (iii) **QUERY**: decide whether there is a valid configuration C for a set of rules satisfying a set of requirements Q ($C \models Q$).

First, we observe that **C-SAT** is decidable in linear time. Second, we note that checking whether a set of atoms is a valid configuration can be done in linear time. This holds as for a set of rules and a candidate configuration, the reduct can be computed in linear time and, similarly, the unique least model of a set of Horn clauses is computable in linear time [4]. This implies that the major computational tasks in configuration using our semantics are in **NP**.

For **EXISTS** and **QUERY**, we consider some subclasses of **CRL** to show the boundary for **NP**-completeness. For example, **CRL_r** is the subset where only requires-rules are allowed, **CRL_{rd}** permits additionally default negations, **CRL_{re}** allows exclusive choice-rules in addition to requires-rules and **CRL_{rcl}** admits requires-rules, choice-rules and incompatibility-rules. The results are summarized in Table 1. They are fairly straightforward to demonstrate (see [17] for more details). Most of the results can also be established from the complexity results for the possible model semantics [14, 5].

Table 1. Complexity results for configuration tasks

Language	C-SAT	EXISTS	QUERY
CRL_r	Poly	Poly	Poly
CRL_{ri}	Poly	Poly	Poly
CRL_{rc}	Poly	Poly	NP-compl.
CRL_{rd}	Poly	NP-compl.	NP-compl.
CRL_{re}	Poly	NP-compl.	NP-compl.
CRL_{rcl}	Poly	NP-compl.	NP-compl.

6 Relation to Constraint Satisfaction

Configuration is often cast as a constraint satisfaction or dynamic constraint satisfaction problem. In this section we aim to show that **CRL** contains CSP and DCSP as special cases and is an extension of these two approaches. We note that for all the formalisms dealt with in this section the problem corresponding to generating a configuration is **NP**-complete.

6.1 Mapping Constraint Formalisms to CRL

We first recall that a CSP consists of a set of *variables*, a set of possible *values* for each variable, called the *domain* of the variable, and a set of *constraints*. We assume in the following that the domains are finite. A constraint defines the allowed combinations of values for a set of variables by specifying a subset of the Cartesian product of the domains of the variables. A *solution* to a CSP is an *assignment* of values to all variables such that the constraints are satisfied, i.e., the value combinations are allowed by at least one tuple of each constraint.

A DCSP is an extension of a CSP that also has of a set of variables, domains, and constraints (called here *compatibility constraints*). However, all the variables need not be given a value, i.e., be *active* in a solution. A DCSP additionally defines a set of *initial variables* that must be active in every solution and a set of *activity constraints*. An activity constraint states either that if a given condition is true then a certain variable is active, or that if a given condition is true, then a certain variable must not be active. The condition may be expressed as a compatibility constraint (*require* and *require not* activity constraints) or it may state that some other variable is active (*always require* and *always require not* activity constraints). A solution to a DCSP is an assignment of values to variables such that it (i) fulfills the compatibility and activity constraints, (ii) contains assignments for the initial variables, and (iii) is minimal.

We next define a mapping from the DCSP formalism to **CRL**. We note that as CSP is a special case of DCSP with no activity constraints and with all variables in the set of initial variables, the same mapping can be used for a CSP. In the mapping from a DCSP to **CRL** representation we introduce (i) a new distinct atom for each variable, v_i , to encode its activity, (ii) a new distinct atom $sat(c_i)$ for each compatibility constraint c_i , and (iii) a new distinct atom $v_i(val_{i,j})$ for each variable v_i and value $val_{i,j}$ in the domain of v_i .

Each initially active variable v_i is mapped to a fact $v_i \leftarrow$. Each variable v_i and its domain $\{val_{i,1}, \dots, val_{i,n}\}$ is mapped to an exclusive choice-rule of the following form: $v_i(val_{i,1}) \oplus \dots \oplus v_i(val_{i,n}) \leftarrow v_i$. A compatibility constraint on variables v_1, \dots, v_n is represented using a set of requires-rules of form $sat(c_i) \leftarrow v_1(val_{1,j}), v_2(val_{2,k}), \dots, v_n(val_{n,l})$, one rule for each allowed value combination $val_{1,j}, \dots, val_{n,l}$. An incompatibility-rule of the form $\leftarrow v_1, \dots, v_n, not(sat(c_i))$ is included to enforce the constraint.

Example 5. Given a CSP with two variables, *package* and *frame* with domains $\{luxury, deluxe, standard\}$ and $\{convertible, sedan, hatchback\}$, respec-

tively, and a constraint $c_1 = \{\{luxury, convertible\}, \{standard, hatchback\}\}$ on *package* and *frame*, the following rule set is produced by the mapping:

```

package ←
frame ←
package(luxury) ⊕ package(deluxe) ⊕ package(standard) ← package
frame(convertible) ⊕ frame(sedan) ⊕ frame(hatchback) ← frame
sat(c1) ← package(luxury), frame(convertible)
sat(c1) ← package(standard), frame(hatchback)
← package, frame, not(sat(c1))
    
```

An always require activity constraint is mapped to a requires-rule $v_2 \leftarrow v_1$ where v_2 is the activated variable and v_1 is the condition variable. An always require not activity constraint is mapped to an incompatibility-rule $\leftarrow v_1, v_2$ where v_1 and v_2 are the condition and deactivated variables, respectively. A require variable activity constraint is mapped to a set of requires-rules, one rule of the form $u \leftarrow v_1(val_{1,j}), \dots, v_n(val_{n,k})$ for each allowed value combination $\{val_{1,j}, \dots, val_{n,k}\}$ of variables v_1, \dots, v_n , where u is the activated variable. A require not activity constraint is mapped to a set of incompatibility-rules, one rule of the form $\leftarrow u, v_1(val_{1,j}), \dots, v_n(val_{n,k})$ for each allowed value combination $\{val_{1,j}, \dots, val_{n,k}\}$ of variables v_1, \dots, v_n where u is the deactivated variable.

Example 6. Given a DCSP with two variables, *package* and *sunroof*, whose domains are $\{luxury, deluxe, standard\}$ and $\{sr_1, sr_2\}$, respectively, a set of initial variables $\{package\}$ and a require activity constraint that if *package* has value *luxury*, then *sunroof* is active, the following rule set is produced:

```

package ←
package(luxury) ⊕ package(deluxe) ⊕ package(standard) ← package
sunroof(sr1) ⊕ sunroof(sr2) ← sunroof
sunroof ← package(luxury)
    
```

It is easy to see that each valid configuration is a solution to the DCSP and vice versa. The minimality of solutions can be shown by noting that the rules that can cause a variable to be active can be translated to normal logic programs. For this subclass of rules the configurations coincide with stable models which are subset minimal [6]. The size of the resulting rule set is linear in the size of the DCSP problem instance. The mapping is *local* in the sense that each variable and its domain, initial variable, compatibility constraint and activity constraint can be mapped separately from the other elements of the problem instance.

6.2 Expressiveness of CRL vs. CSP

Next we argue that **CRL** is strictly more expressive than **CSP** by using the concept of *modularity*. A modular representation in some formalism is such that a small, local change in the knowledge results in a small change in the representation. This property is important for easy maintenance of a knowledge base.

We show that under mild assumptions the CSP formalism cannot modularly capture the justifications of a configuration. We say that **CRL** is *modularly representable* by CSP iff for every set of **CRL** rules there is a CSP such that rules are represented in the CSP independent of the representation of the basic facts (i.e. requires-rules with empty bodies) so that a change in the facts does not lead to a change involving both additions and removals of either allowed tuples, constraints, variables or values. In addition, the solutions to the CSP must *agree* with the **CRL** configurations in that (i) the truth values of the atoms in a configuration can be read from the values of Boolean CSP variables representing the atoms and (ii) these variables have the same truth values as the corresponding atoms.

Theorem 1. ***CRL** is not modularly representable by CSP.*

Proof. Consider the set of rules $R = \{c \leftarrow b\}$ and assume that it can be modularly represented by a CSP. Hence, there is a CSP $T(R)$ such that in all the solutions of $T(R)$ the variables representing atoms b and c in the configuration language have the value *false* as R has the empty set as its unique valid configuration. Consider now a set of facts $F = \{b \leftarrow\}$. The configuration model $R \cup F$ has a unique valid configuration $\{b, c\}$. This means that $T(R)$ updated with F must not have a solution in which variables encoding b and c have the value *false*. In addition, $T(R)$ updated with F must have at least one solution in which the atoms encoding b and c have the value *true*. It can be shown that changes including either only additions or only removals of either allowed tuples, constraints, variables or values cannot both add solutions and remove them, which is a contradiction and hence the assumption is false.

The fact that there is no modular representation of **CRL** in the CSP formalism is caused by the justification property of **CRL** which introduces a non-monotonic behavior. A similar argument can therefore be used for showing a similar result for, e.g., propositional logic [17]. We note that the question whether there is a modular representation of a configuration model given in **CRL** as a DCSP is open. The DCSP formalism exhibits a non-monotonic behavior, so a similar argument cannot be used for this case. It can be used, however, to show that there is no modular representation of a DCSP as a CSP. Representing **CRL** as DCSP does not seem straightforward, as the DCSP approach does not directly allow activity constraints that have a choice among a set of variables to activate or default negation in the condition part.

7 Implementation

In this section we describe briefly our implementation of **CRL**, demonstrate the use of **CRL** with a car configuration problem from [11] and provide information on performance of the implementation for the car problem.

Our implementation of **CRL** is based on the translation of **CRL** to normal logic programs presented in Sect. 4 and on an existing high performance implementation of the stable model semantics, the Smodels system [12,13]. This

system seems to be the most efficient implementation of the stable model semantics currently available. It is capable of handling large programs, i.e. over 100 000 ground rules, and has been applied successfully in a number of areas including planning [2], model checking for distributed systems [9], and propositional satisfiability checking [16].

We have built a front-end to Smodels which takes as input a slightly modified (see below) set of **CRL** rules and transforms it to a normal logic program whose stable models correspond to valid configurations. Then Smodels is employed for generating stable models. The implementation can generate a given number of configurations, all of them, or the configurations that satisfy requirements given as a set of literals.

Smodels is publicly available at <http://www.tcs.hut.fi/pub/smodels/>. The front-end is included in the new parser of Smodels, `lparse`, which accepts in addition to normal program rules (requires-rules) also “inclusive” choice-rules and incompatibility-rules. Exclusive choice-rules are supported by rules of the form $\leftarrow n\{a_1, \dots, a_l\}$ where n is an integer. The rule acts like an integrity constraint eliminating models, i.e. configurations, with n or more of the atoms from $\{a_1, \dots, a_l\}$. This allows a succinct coding of, e.g., exclusiveness without the quadratic overhead which results when using normal rules. Hence, an exclusive choice-rule $a_1 \oplus \dots \oplus a_l \leftarrow Body$ can be expressed as a combination of an “inclusive” choice-rule $a_1 \mid \dots \mid a_l \leftarrow Body$ and the rule $\leftarrow Body, 2\{a_1, \dots, a_l\}$.

Our first example, **CAR**, was originally defined as a DCSP [11]. In Fig. 11 the problem is translated to **CRL** using the mappings defined in the previous section with the exception that the compatibility constraints are given a simple rule form similar to that in [11]. There are several choices of packages, frames, engines, batteries and so on for a car. At least a package (*pack*), frame and engine must be chosen from the alternatives specified for them. Choosing a particular alternative in a choice-rule can make other choices necessary. For example, if the package is chosen to be luxury (*l*), then a sunroof and an airconditioner (*aircond*) must be chosen as well. In addition, some combinations of alternatives are mutually exclusive, e.g., the luxury alternative for package cannot be chosen with the *ac1* alternative for airconditioner. The second example, **CARx2**, is modified from **CAR** by doubling the size of the domain of each variable. In addition, for each new value and each compatibility and activity constraint in the original example a new similar constraint referring to the new value is added.

We did some experiments with the two problems in **CRL** form. The tests were run on a Pentium II 233 MHz with 128MB of memory, Linux 2.0.35 operating system, `smodels` version 1.12 and `lparse` version 0.9.19. The test cases are available at <http://www.tcs.hut.fi/pub/smodels/tests/pad199.tar.gz>. Table 2 presents the timing results for computing one and all valid configurations, the number of valid configurations found and the size of the initial search space which is calculated by multiplying the number of alternatives for each choice. The execution times include reading and parsing the set of input rules, its translation to a normal program as well as outputting the configurations in a file. The times were measured using the Unix `time` command and they are the sum of

$pack(l) \oplus pack(dl) \oplus pack(std) \leftarrow pack$ $frame(conv) \oplus frame(sedan) \oplus frame(hb) \leftarrow frame$ $engine(s) \oplus engine(m) \oplus engine(l) \leftarrow engine$ $battery(s) \oplus battery(m) \oplus battery(l) \leftarrow battery$ $sunroof(sr1) \oplus sunroof(sr2) \leftarrow sunroof$ $aircond(ac1) \oplus aircond(ac2) \leftarrow aircond$ $glass(tinted) \oplus glass(nottinted) \leftarrow glass$ $opener(auto) \oplus opener(manual) \leftarrow opener$ $battery(m) \leftarrow opener(auto), aircond(ac1)$ $battery(l) \leftarrow opener(auto), aircond(ac2)$ $\leftarrow sunroof(sr1), aircond(ac2), glass(tinted)$ $\leftarrow pack(std), aircond(ac2)$ $\leftarrow pack(l), aircond(ac1)$ $\leftarrow pack(std), frame(conv)$	$pack \leftarrow$ $frame \leftarrow$ $engine \leftarrow$ $sunroof \leftarrow pack(l)$ $aircond \leftarrow pack(l)$ $sunroof \leftarrow pack(dl)$ $opener \leftarrow sunroof(sr2)$ $aircond \leftarrow sunroof(sr1)$ $glass \leftarrow sunroof$ $battery \leftarrow engine$ $sunroof \leftarrow opener$ $sunroof \leftarrow glass$ $\leftarrow sunroof(sr1), opener$ $\leftarrow frame(conv), sunroof$ $\leftarrow battery(s), engine(s),$ $aircond$
--	--

Fig. 1. Car configuration example

user and system time. The test results show that for this small problem instance the computation times are acceptable for interactive applications. For example, in the larger test case it takes on average less than 0.0004 s to generate a configuration. We are not aware of any other reported test results for solving this problem in the DCSP or any other form.

Table 2. Results from the car example

Problem	Initial search space	Valid configurations	one	all
CAR	1 296	198	0.06 s	0.15 s
CARx2	331 776	44456	0.1 s	15.5 s

8 Previous Work on Product Configuration

In Sect. 6 we compared our approach to the CSP and DCSP formalisms. In this section we provide brief comparisons with several other approaches.

The generative CSP (GCSP) [7] approach introduces first-order constraints on activities of variables, on variable values and on resources. Constraints using arithmetic are also included. *Resources* are aggregate functions on intensionally defined sets of variables. They may restrict the set of variables active in a solution or generate new variables into a solution, thus providing a justification for the variables. In addition, a restricted form of DCSP activity constraints is used to provide justifications for activity of variables. **CRL** allows more expressive

activity constraints than DCSP and a uniform representation of activity and other constraints. However, first-order rules, arithmetic and resource constraints are still missing from **CRL**.

Our approach fits broadly within the framework of constructive problem solving (CPS) [8]. In CPS the configurations are characterized as (possibly partial) Herbrand models of a theory in an appropriate logic language. The CPS approach does not require that elements in a configuration must have justifications but the need for a meta-level minimality criterion is mentioned.

Some implementations of configurators based on logic programming systems have been presented [15, 1]. In these approaches, similarly to our approach, a configuration domain oriented language is defined and the problem solving task is implemented on a variant of Prolog based on a mapping from the high-level language to Prolog. The languages are more complex and better suited for real modeling tasks. However, they are not provided a clear declarative semantics and the implementations use non-logical extensions of pure Prolog such as object-oriented Prolog and the cut. In contrast, we provide a simple declarative semantics and a sound and complete implementation for **CRL**.

9 Conclusions and Future Work

We have defined a rule-based language for representing typical forms of configuration knowledge, e.g., choices, dependencies between choices and incompatibilities. The language is provided with a declarative semantics based on a straightforward fix-point condition employing a simple transformation operator. The semantics induces formal definitions for the main concepts in product configuration, i.e., configuration models, requirements, configurations, valid configurations and configurations that satisfy requirements. A novel feature of the semantics is that justifiability of a configuration (i.e., that each element in a configuration has a justification in terms of the configuration rules) is captured by Horn clause derivability but without resorting to a minimality condition on configurations. This approach has not been considered in previous work on product configuration. The semantics is closely related to well-known non-monotonic formalisms such as the stable model semantics [6] and the possible model semantics [14].

Avoiding minimality conditions in the semantics has a favorable effect on the complexity of the configuration tasks. The basic problems, i.e. validity of a configuration and whether a configuration satisfies a set of requirements, are polynomially decidable. This is important for practical configuration problems. It also implies that the other relevant decision problems are in **NP**.

We argue that the rule language is more expressive than constraints by showing that it cannot be modularly represented as CSP. The difficulty lies in capturing the justifications for a configuration using constraints. In addition, we show that the dynamic constraint satisfaction formalism can be embedded in our language but note that there is no obvious way of representing default negation and inclusive choices of **CRL** in that formalism.

There are indications that the proposed formal model provides a basis for solving practically relevant product configuration problems. An implementation of the rule language based on a translator to normal logic programs with the stable model semantics was tested on a small configuration problem. The results suggest that this approach is worth further research. Moreover, experiences in other domains show that efficient implementations of the stable model semantics are capable of handling tens of thousands of ground rules. Compiling a practically relevant configuration model from a high level representation into our language would seem to generate rule sets of approximately that size. Further research is needed to determine how our implementation scales for larger problems.

It may be possible to develop a more efficient algorithm that avoids the overhead incurred by the additional atoms and loss of information on the structure of the rules caused by the mapping to normal programs. Devising such an algorithm is an interesting subject of further work. A practically important task would be to identify additional syntactically restricted but still useful subsets of the language that would allow more efficient computation. Interactive product configuration where user makes hard decisions and computer only tractable ones may be the only feasible alternative for very large or complex problems. This type of configuration would be facilitated by devising polynomially computable approximations for valid configurations in **CRL**. Such approximations could also be used to prune the search space in an implementation of **CRL**.

It should be noted that the model does not adequately cover all the aspects of product configuration. Further work should include generalizing the rules to the first-order case, adding arithmetic operators to the language and defining constructs important for the domain such as optional choice directly in the language. These extensions are needed to conveniently represent resource constraints, attributes, structure and connections of components. Another important extension would be to define the notion of an optimal configuration (such as subset minimal, cardinality minimal or resource minimal configuration) and to analyze the complexity of optimality-related decision problems.

Acknowledgements. The work of the first author has been supported by the Helsinki Graduate School in Computer Science and Engineering (HeCSE) and the Technology Development Centre Finland and the work of the second author by the Academy of Finland through Project 43963. We thank Tommi Syrjänen for implementing the translation of **CRL** to normal logic programs.

References

1. T. Axling and S. Haridi. A tool for developing interactive configuration applications. *Journal of Logic Programming*, 19:658–679, 1994.
2. Y. Dimopoulos, B. Nebel, and J. Koehler. Encoding planning problems in non-monotonic logic programs. In *Proceedings of the Fourth European Conference on Planning*. Springer-Verlag, 1997.

3. J. Dix. Semantics of logic programs: Their intuitions and formal properties. In *Logic, Action and Information — Essays on Logic in Philosophy and Artificial Intelligence*, pages 241–327. DeGruyter, 1995.
4. W.F. Dowling and J.H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 3:267–284, 1984.
5. T. Eiter and G. Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Mathematics and Artificial Intelligence*, 15:289–323, 1995.
6. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080. The MIT Press, 1988.
7. A. Haselböck and M. Stumptner. An integrated approach for modelling complex configuration domains. In *Proceedings of the 13th International Conference on Expert Systems, AI, and Natural Language*, 1993.
8. R. Klein. A logic-based description of configuration: the constructive problem solving approach. In *Configuration—Papers from the 1996 AAAI Fall Symposium. Technical Report FS-96-03*, pages 111–118. AAAI Press, 1996.
9. X. Liu, C Ramakrishnan, and S. Smolka. Fully local and efficient evaluation of alternating fixed points. In *Proceedings of 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 5–19. Springer-Verlag, 1998.
10. J. McDermott. R1: a rule-based configurer of computer systems. *Artificial Intelligence*, 19(1):39–88, 1982.
11. S. Mittal and B. Falkenhainer. Dynamic constraint satisfaction problems. In *Proc. of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, pages 25–32. AAAI, MIT Press, 1990.
12. I. Niemelä and P. Simons. Efficient implementation of the well-founded and stable model semantics. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 289–303. The MIT Press, 1996.
13. I. Niemelä and P. Simons. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning*, pages 420–429. Springer-Verlag, 1997.
14. C. Sakama and K. Inoue. An alternative approach to the semantics of disjunctive logic programs and deductive databases. *Journal of Automated Reasoning*, 13:145–172, 1994.
15. D. Searls and L. Norton. Logic-based configuration with a semantic network. *Journal of Logic Programming*, 8(1):53–73, 1990.
16. P. Simons. Towards constraint satisfaction through logic programs and the stable model semantics. Research report A47, Helsinki University of Technology, Helsinki, Finland, 1997. Available at <http://saturn.hut.fi/pub/reports/A47.ps.gz>.
17. T. Soininen and I. Niemelä. Formalizing configuration knowledge using rules with choices. Research report TKO-B142, Helsinki University of Technology, Helsinki, Finland, 1998. Presented at the Seventh International Workshop on Nonmonotonic Reasoning (NM'98), 1998.
18. J. Tiihonen, T. Soininen, T. Männistö, and R. Sulonen. State-of-the-practice in product configuration—a survey of 10 cases in the Finnish industry. In *Knowledge Intensive CAD*, volume 1, pages 95–114. Chapman & Hall, 1996.
19. E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London, 1993.

University Timetabling Using Constraint Logic Programming

Hans-Joachim Goltz and Dirk Matzke

GMD – German National Research Center for Information Technology
GMD-FIRST, Rudower Chaussee 5, D-12489 Berlin
`goltz@first.gmd.de`

Abstract. A timetable is a temporal arrangement of a set of meetings such that all given constraints are satisfied. A timetabling problem can be suitably modelled in terms of a set of constraints. We use Constraint Logic Programming and develop methods, techniques and concepts for a combination of interactive and automatic timetabling of university courses and school curricula. An exemplary application of such a timetabling system was developed for the Charité Medical Faculty at the Humboldt University, Berlin. The timetabling system is flexible enough to take into account special user requirements and to allow constraints to be modified easily if no basic conceptual change in the timetabling is necessary. An essential component is an automatic heuristic solution search with an interactive user-intervention facility. The user will, however, only be able to alter a timetable or schedule such that no hard constraints are violated.

1 Introduction

Constraint Logic Programming over finite domains is a rapidly growing research field aiming at the solution of large combinatorial problems. For many real-life problems, the Constraint Logic Programming approach has been applied with great success. A timetabling problem can be defined as the scheduling of a certain number of courses that have to satisfy specific conditions, in particular constraints regarding demands on limited resources. Besides the so-called hard constraints, all of which must be satisfied, there are also constraints which should be satisfied as far as possible; these are termed soft constraints. A timetabling problem can be suitably modelled in terms of a set of constraints. Constraint Logic Programming allows formulation of all constraints in a declarative manner.

Timetabling has long been known to belong to the class of problems called NP-complete (see e.g. [9]). Existing software products for timetabling impose too severe restrictions on the kind of timetables that can be generated. In addition, they are not flexible enough to allow anomalous requirements to be integrated or the special features found in virtually every user organization to be taken into account (see e.g. [7]). Automated timetabling is a current and relevant field of research.

The Second International Conference on Automated Timetabling was held in 1997 ([8,6]). Different software methods and approaches are available for automated timetabling (see e.g. [19]). Various Constraint Logic Programming approaches are discussed in [13,5,13,15,17,16,20].

In [3], the Constraint Logic Programming language DOMLOG is used to solve the timetabling problem of a computer science department. This language contains user-defined heuristics for search methods and allows greater flexibility in the declaration of “forward” and “look-ahead” constraints (w.r.t. other such languages). The timetabling problems of a computer science department are also addressed in [16] and [1], the former using the Constraint Logic Programming language ECLⁱPS^e and the latter selecting the high-level constraint language Constraint Handling Rules (CHR) as the implementation language. The language CHR was developed by Tom Frühwirth [11]. Based on an existing domain solver written in CHR, a solver for the timetabling problem was developed, requiring no more than 20 lines of code. [5] and [13] look at the application of the Constraint Logic Programming language CHIP to a university’s examination timetabling and describe the use of the global constraint *cumulative*. Different labelling strategies are discussed in [13].

M. Henz and J. Würtz [15] present a solution to the timetabling problem of a German college using the language Oz. This language was developed by the DFKI at Saarbrücken (Germany) and is a concurrent language allowing for functional, object-oriented and constraint programming. G. Lajos [17] describes experience in constructing a large-scale modular timetable using Constraint Logic Programming. Arithmetic constraints and the built-in constraint *atmost* are only used for problem modelling. In [20], a hybrid approach for the automatic generation of university and college timetables is presented. This approach combines features of constraint logic and tabu search. These two approaches are also discussed separately.

Our research is concerned with the development of methods, techniques and concepts for a combination of interactive and automatic timetabling of university courses and school curricula. The automated solution search will be implemented in such a way that, it normally allows a solution to be found in a relatively short time, if such a solution exists. The timetabling systems will be flexible enough to take into account special user requirements and to allow constraints to be modified easily, if no basic conceptual change in the timetabling is necessary. An essential component is an automatic heuristic solution search with an interactive user-intervention facility. The user will, however, only be able to alter a timetable or schedule such that no hard constraints are violated.

The research focuses on the following areas: concepts and methods for modelling timetabling problems; study of the influence of different modelling techniques on the solution search; development and comparative analysis of different methods and techniques for heuristic solution search; interactive user intervention in the solution search via a graphical interface. The goals are to be met using Constraint Logic Programming, this being the approach best suited to our

needs. The methods, techniques and concepts developed or under development are tested and further elaborated on prototypical applications.

An exemplary application of a combined interactive and automatic course-timetabling system was developed for the Charité Medical Faculty at the Humboldt University, Berlin. This system is called *CharPlan* and realizes timetabling by means of Constraint Logic Programming and a special algorithm for an efficient search. The first test phase has been successfully completed. The generated timetables were output as HTML files and are available on the Internet.

In this paper, we describe the generation of this timetabling system. Section 2 gives a brief description of the timetabling problem. This is followed by a discussion of Constraint Logic Programming, problem representation and search methods. Some remarks on the implementation and results are also given.

2 Problem Description

Students of medicine have to attend courses covering all medical fields. There are different types of courses: lectures, seminars and practicals. The seminars and practicals are held in groups of up to 20 students. Each student in a particular semester participates in one of these groups, there being between 12 and 24 groups per semester. The buildings of the Charité Medical Faculty at the Humboldt University are at two different locations in Berlin. Some practicals are also carried out in clinics and hospitals in other parts of Berlin. Other important constraints for this timetabling problem are:

- C_1 : The courses can begin at every quarter hour and may run for different lengths of time.
- C_2 : There are restrictions with respect to the starting time for each course.
- C_3 : Two lectures on a same subject may not be scheduled for the same day.
- C_4 : Each group's lectures, seminars and practicals of each group should not overlap.
- C_5 : The number of seminars and practicals a department (research group) held at any one time is limited.
- C_6 : Some courses are only held during certain weeks, the following arrangements are possible: every week, A-weeks (weeks with odd numbers: 1,3,5, ...), B-weeks (weeks with even numbers: 2,4,6, ...), either A-weeks or B-weeks.
- C_7 : Lectures on the same subject are carried out in parallel at different parts of the faculty. Thus a student can choose which of the parallel lectures to attend.
- C_8 : Timetabling also involves allocating rooms for the individual lectures.
- C_9 : Some lectures can only be held in certain specified rooms; in some cases only one room matches the requirements.
- C_{10} : The varying distances between the different course locations must be taken into account.
- C_{11} : The starting time preferences for courses and the preferred rooms for lectures should also be taken into account where possible.

The constraint C_{11} is a soft constraint. The other constraints are hard constraints and must be satisfied.

3 Constraint Logic Programming

Constraint Logic Programming (CLP) is a generalization of Logic Programming, unification being replaced by constraint handling in a constraint system. CLP combines the declarativity of Logic Programming with the efficiency of constraint-solving algorithms. In the Constraint Logic Programming paradigm, a constraint satisfaction problem can be represented by Horn clause logic programs in which the clause bodies may contain constraints. Constraints are generated incrementally during runtime and passed to a constraint solver which applies domain-dependent constraint satisfaction techniques to find a feasible solution for the constraints. Constraint Logic Programming with constraints over finite integer domains, CLP(FD), has been established as a practical tool for solving discrete combinatorial problems.

The Constraint Logic Programming language CHIP ([10]) was one of the first CLP languages, together with PROLOG III and CLP(\mathcal{R}). It was developed at the European Computer-Industry Research Centre (ECRC) in Munich between 1985 and 1990, and is now being developed and marketed by the French firm COSYTEC. The current version of CHIP differs in particular from the earlier constraint systems by the inclusion of global constraints ([24]). The CHIP system contains different constraint solvers. We only use constraints over finite domains. Some built-in constraints of the CLP(FD) part of CHIP are briefly described below.

Each constrained variable has a domain that must first be defined. Such a variable is also called domain variable. The usual arithmetic constraints *equality*, *disequality*, and *inequalities* can be applied over linear terms built upon domain variables and natural numbers. The *alldifferent* constraint is a symbolic constraint and expresses a constraint that all members of a list of domain variables and integers must have different values. Instead of this constraint, a logical equivalent conjunction of disequalities could be used. The disadvantage of using a conjunction of disequalities is that consistency methods for such a conjunction are quite weak as the disequalities are considered in isolation. Symbolic constraints can be used for more complex conditions concerning the relation between several domain variables. A change in the domain of one variable may have an effect on all other variables occurring in the constraint. A very useful symbolic constraint is `element(N,List,Value)`. It specifies that the N^{th} element of the nonempty list `List` of natural numbers must have the value `Value`, where `Value` is either a natural number, a domain variable or a free variable.

Global constraints use domain-specific knowledge to obtain better propagations results and can be applied to large problem instances. Complex conditions on sets of variables can be modelled declaratively by such constraints and can be used in multiple contexts. Global constraints with specialized consistency methods can greatly improve efficiency for solving real-life problems. The basic ideas of the global constraints *diffn* and *cumulative* are given in the rest of this section.

The *cumulative constraint* was originally introduced to solve scheduling and placement problems ([2]). The simplified form of this constraint is

$\text{cumulative}([S_1, S_2, \dots, S_n], [D_1, D_2, \dots, D_n], [R_1, R_2, \dots, R_n], L),$

where $[S_1, S_2, \dots, S_n]$, $[D_1, D_2, \dots, D_n]$, and $[R_1, R_2, \dots, R_n]$ are nonempty lists of domain variables (or natural numbers) and L is a natural number. The usual interpretation of this constraint is a single resource scheduling problem:

Each S_i represents the starting time of some event, D_i is the duration and R_i the amount of resources needed by this event. L is the total amount of resources available at each relevant point of time. The cumulative constraint ensures that, at each time point in the schedule, the amount of resources consumed does not exceed the given limit L .

The mathematical interpretation of this constraint is:

for each $k \in [\min\{S_i\}, \max\{S_i + D_i\} - 1]$ it holds that:
 $\sum R_j \leq L$ for all j with $S_j \leq k \leq S_j + D_j - 1$

The *diffn constraint* was included in CHIP to handle multidimensional placement problems. The basic diffn constraint takes as arguments a list of n-dimensional rectangles, where origin and length can be domain variables with respect to each dimension. An n-dimensional rectangle is represented by a tuple $[X_1, \dots, X_n, L_1, \dots, L_n]$, where X_i is the origin and L_i the length of the rectangle in the i^{th} dimension. This constraint ensures that the given n-dimensional rectangles do not overlap. Figure 1 gives an example of three two-dimensional rectangles with the nonoverlapping constraint. The largest rectangle, for instance, is determined by the second list $[1, 2, 3, 2]$: the origin is the point $(1, 2)$, and the lengths of the two dimensions are 3 and 2.

Special conditions of n-dimensional rectangles can also be expressed by diffn constraints. The consistency methods for these constraints are very complex because there are numerous possibilities for inferring new information, depending on which of the variables are known.

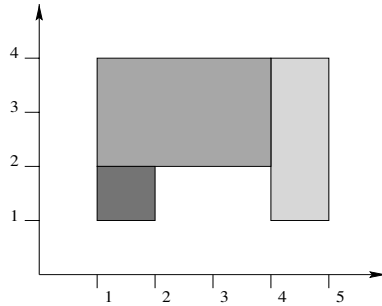


Fig. 1. $\text{diffn}([1, 1, 1, 1], [1, 2, 3, 2], [4, 1, 1, 3])$

4 Problem Representation

A timetabling problem can be suitably modelled in terms of a set of constraints. Since constraints are used, the problem representation is declarative. In this section, the problem representation is described briefly. The method chosen for problem representation has a considerable influence on the solution search. The success of the search often depends directly on the model chosen. Furthermore, the modelling options depend on the chosen constraint solver. We use the constraint solver over finite domains of the Constraint Logic Programming language CHIP. The global constraints built into CHIP are particularly useful for problem modelling.

For each course the *starting time* is represented by a domain variable. If we assume that a course can be held on five days of a certain week between 8 a.m. and 8 p.m., then $5 \times 12 \times 4 = 240$ time units have to be considered. This means, initially, that the domains for the starting time variables are defined by the natural numbers $1, 2, \dots, 240$.

The attributes *room*, *week* and *location* of a course may be domain variables if these values are unknown. The possible values of these attributes are mapped into the natural numbers.

With the exception of the constraint C_{11} , all other constraints of the timetabling problem are directly represented by built-in constraints. C_{11} is a soft constraint and is integrated into the solution search. Conditional constraints are not used.

The constraints C_1 and C_2 can be easily satisfied by the choice of the domains for the variables of *starting time* and by corresponding restrictions of these domains. Analogously, the constraint C_9 can be satisfied. Note that the capacity of a room is taken into account by the choice of the domain of a room variable.

If two lectures are not scheduled for the same day, then one lecture is scheduled at least one day before the other lecture. Since two lectures on the same subject have the same features, we can determine which lecture is scheduled at least one day before the other. For example, assume that $S1 \text{ in } 1..240$ and $S2 \text{ in } 1..240$ are the domain variables for the starting times of two such lectures. Furthermore, let $X \text{ in } [48, 96, 144, 192]$ be a new domain variable, where $\{48, 96, 144, 192\}$ are the last time units of the first four days. Then, the relations $S1 \#< X$ and $X \#< S2$ ensure the constraint C_3 . This method of modelling is more efficient than the method suggested in [13] for the same constraint. In [13], for each lecture, the domain variables $Day \text{ in } 1..5$ and $Hour \text{ in } 1..48$ are considered additionally (X is not considered) and the constraint $S \# = 48 * (Day - 1) + Hour$ is generated. Then, the constraint C_3 is modelled using the predicate *alldifferent* with respect to the *Day*-variables. The following example shows the different propagation properties. If we consider the question

¹ The arithmetic relations of the finite domain solver are indicated by $\#$ in CHIP

```
?- Day1 in 1..5, Hour1 in 1..48, S1 in 1..240,
    S1 + 48 #= 48*Day1 + Hour1,
    Day2 in 1..5, Hour2 in 1..48, S2 in 1..240,
    S2 + 48 #= 48*Day2 + Hour2,
    alldifferent([Day1,Day2]),
    Day1 = 3.
```

we get the answer:

```
Day1 = 3
Hour1 = Hour1 in {1..48}
S1 = S1 in {97..144}
Day2 = Day2 in {1..2,4..5}
Hour2 = Hour2 in {1..48}
S2 = S2 in {1..240}
```

The corresponding question for our method is:

```
?- S1 in 1..240, S2 in 1..240,
    X in [48,96,144,192],
    S1 #<= X, X #< S2,
    96 #< S1, S1 #<= 144.
```

In this case, $\text{Day1} = 3$ is modelled by the two constraints $96 \#< S1$ and $S1 \#<= 144$. We get the following answer to this question:

```
S1 = S1 in {97..144}
S2 = S2 in {145..240}
X = X in {144,192}
```

With our method, the domain of the variable $S2$ is reduced more than with the method suggested in [13]. However, our method can only be used if the two lectures have the same features. The method given in [13] can also be used in other cases. We can extend our method to cover the case that the durations ($d1$, $d2$) of the lectures are different. In this case, domain variables are defined for duration: $D1$ in $[d1,d2]$ and $D2$ in $[d1,d2]$. These variables are linked by the constraint $D1 + D2 \# = d1 + d2$.

The constraints C_4 and C_5 can be represented by cumulative constraints. For example, let $S1, S2, \dots, S_n$ be the domain variables of starting times of the courses held by a certain department and let $D1, D2, \dots, D_n$ be the corresponding lengths of time. If Max is the maximum number of courses that this department can give at any one time, then the constraint C_5 can be modelled by²:

```
cumulative([S1,S2,...,Sn], [D1,D2,...,Dn], [1,1,...,1], Max)
```

For each group, the constraint C_4 can be modelled analogously, where Max being equal to 1 in this case.

² The unused arguments are not mentioned.

The use of *diffn* constraints can ensure that the constraints C_6 and C_7 are satisfied, a course being represented by a "3-dimensional rectangle" with the dimensions *starting time*, *week* and *parallelism*. If the allocation of rooms is required, then the attribute *room* has to be considered as a fourth dimension (C_8). In the following examples, we assume that *week* and *parallelism* do not need to be considered. Let S_1, S_2, \dots, S_n be the domain variables of starting times, D_1, D_2, \dots, D_n be the corresponding durations, and let R_1, R_2, \dots, R_n be the domain variables for room allocations. The constraint C_8 can be modelled by the following constraint, where the length of the dimension "room" is 1 unit:

`diffn([[S1,D1,R1,1], [S2,D2,R2,1], ..., [Sn,Dn,Rn,1]])`

This constraint can also be modelled by the cumulative constraint, where additional domain variables X_1, X_2, \dots, X_n are required. For each i , the variables S_i, R_i, X_i are linked by the equation $X_i = N * R_i + S_i$, where N is greater than or equal to the maximum number of time units. A similar method of modelling with cumulative constraints is discussed in [5].

We consider a simple example of these two methods: 4 lectures (the first four) with a duration of 2 time units, 2 lectures with a duration of 1 time unit, and two rooms (1 and 2). The lectures are scheduled within 6 time units, and the second and the fifth lectures cannot be allocated to room 1. Furthermore, we assume that the first lecture is scheduled at time point 3 in room 2. This example can be modelled with the *diffn* constraint as follows:

```
?- [S1,S2,S3,S4] in 1..5, [S5,S6] in 1..6,
    [R1,R2,R3,R4,R5,R6] in 1..2, R2 #\= 1, R5 #\= 1,
    diffn([ [S1,R1,2,1], [S2,R2,2,1], [S3,R3,2,1],
            [S4,R4,2,1], [S5,R5,1,1], [S6,R6,1,1] ]),
    R1=2,S1=3.
```

The answer to this question is:

S1 = 3,	R1 = 2,
S2 = S2 in {1,5},	R2 = 2,
S3 = S3 in {1..5},	R3 = 1,
S4 = S4 in {1..5},	R4 = 1,
S5 = S5 in {1..2,5..6},	R5 = 2,
S6 = S6 in {1..6},	R6 = R6 in {1..2}

If the cumulative constraint³ is used, then this question can be modelled by:

```
?- [S1,S2,S3,S4] in 1..5, [S5,S6] in 1..6,
    [R1,R2,R3,R4,R5,R6] in 1..2, R2 #\= 1, R5 #\= 1,
    [X1,X2,X3,X4,X5,X6] in 11..26,
    X1 #= 10*R1 + S1, X2 #= 10*R2 + S2,
    X3 #= 10*R3 + S3, X4 #= 10*R4 + S4,
    X5 #= 10*R5 + S5, X6 #= 10*R6 + S6,
    cumulative([X1,X2,X3,X4,X5,X6], [2,2,2,2,1,1], [1,1,1,1,1,1], 1),
    X1 = 23.
```

³ The unused arguments are not mentioned.

In this case, the answer to this question is:

S1 = 3,	R1 = 2,
S2 = S2 in {1..5},	R2 = 2,
S3 = S3 in {1..5},	R3 = R3 in {1..2},
S4 = S4 in {1..5},	R4 = R4 in {1..2},
S5 = S5 in {1..6},	R5 = 2,
S6 = S6 in {1..6},	R6 = R6 in {1..2},
X1 = 23,	X2 = X2 in {21,25},
X3 = X3 in {11..21,25},	X4 = X4 in {11..21,25},
X5 = X5 in {21..22,25..26},	X6 = X6 in {11..22,25..26}

The domains of the variables are more reduced if the conditions are defined using the `diffn` constraint. In particular, the constraint solver then determines that `R3` and `R4` can only be equal to 1.

The `diffn` constraints can also be used for constraint `C10`, where at least the dimensions *starting time* and *location* are used. Breaks are needed between courses. The length of a break depends on the distance between locations. These breaks are considered as dummy courses. Such a dummy course is defined for each course and each location. The starting time of a dummy course has to be equal to the finishing time of the corresponding course. The duration of a dummy course is defined by the symbolic built-in constraint *element* and depends on the selected location of the corresponding course.

We consider a simple example of this kind of modelling. Let *A* and *B* be two lectures that can be held at the two locations 1 and 2. To allow for the distance between different locations, a break of 3 time units must be scheduled, and for the same location, a break of 1 time unit is sufficient. The duration of these lectures is assumed to be 4 time units. Let *Sa*, *Sb* be the variables for the starting time, and *La*, *Lb* the variables for the location of these lectures. In this example, we assume that lecture *B* is scheduled after lecture *A*, and we consider only the dummy courses *A1* and *A2* for the breaks needed after lecture *A*. The variables for the starting times of these dummy courses are denoted by *S1* and *S2*, respectively. *D1* and *D2* denote the variables for the duration of these activities. Then, the constraints can be implemented by:

```
example(Sa,La,Sb,Lb) :-
    [Sa,Sb] in 1..20,
    [S1,S2] in 1..20,
    [La,Lb] in 1..2,
    element(La,[1,3],D1),
    element(La,[3,1],D2),
    Sa + 4 #= S1,  Sa + 4 #= S2,
    diffn([Sa,La,4,1],[S1,1,D1,1],[S2,2,D2,1],[Sb,Lb,4,1]),
    Sa + 4 #<= Sb.
```

We assume that lecture *A* is scheduled at time point 1. The following questions demonstrate the interdependence of the variable *Sb* and the choice of the locations *La*, *Lb*.

```
?- example(Sa,La,Sb,Lb), Sa=1,La=1,Lb=1.    ==> Sb = Sb in {6..20}
?- example(Sa,La,Sb,Lb), Sa=1,La=1,Lb=2.    ==> Sb = Sb in {8..20}
?- example(Sa,La,Sb,Lb), Sa=1,La=1,Sb=6.    ==> Lb = 1
?- example(Sa,La,Sb,Lb), Sa=1,La=2,Lb=1.    ==> Sb = Sb in {8..20}
?- example(Sa,La,Sb,Lb), Sa=1,La=2,Lb=2.    ==> Sb = Sb in {6..20}
?- example(Sa,La,Sb,Lb), Sa=1,La=2,Sb=7.    ==> Lb = 2
?- example(Sa,La,Sb,Lb), Sa=1,Sb=7,Lb=2.    ==> La = 2
```

Figure 2 shows the schedule of lecture *A* and the dummy courses for the two different location choices (*La* and *Lb*, respectively).

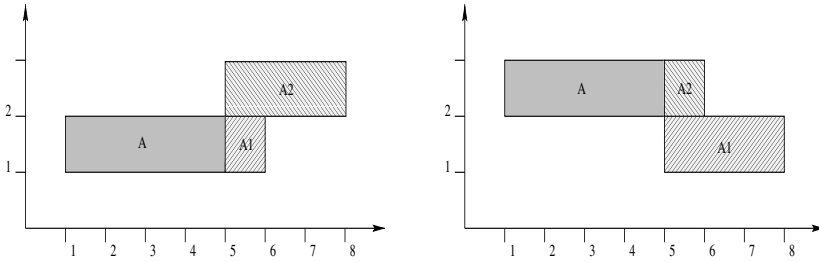


Fig. 2. Lecture *A* with dummy courses for the breaks

5 Search Methods

A solution of a timetabling problem is a scheduling of the given courses such that all hard constraints are satisfied. The soft constraints should be satisfied as far as possible. A constraint solver over finite domains is not complete because consistency is only proved locally. Thus, a search is generally necessary to find a solution. Often, this search is called “labelling”. The basic idea behind this procedure is:

- select a variable from the set of problem variables considered, choose a value from the domain of this variable and assign this value to the variable
- consistency techniques are used by the constraint solver in order to compute the consequences of this assignment; backtracking must be used if the constraint solver detects a contradiction

- repeat this until all problem variables have a value and the constraints are satisfied

We propose a generalization of the labelling method presented in [12]. The assignment of a value to the selected variable is replaced by reduction of the domain of this variable. If backtracking occurs, the unused part of the domain is taken as the new domain for repeated application of this method. The following program describes the basic algorithm of this search:

```
reducing( [], _ ).
reducing(VarList, InfoList) :-
    select_var(Var, VarList, NewVarList),
    reducing_domain(Var, InfoList),
    reducing(NewVarList, InfoList).
```

Domain reduction of a selected variable may be dependent on different components. Thus, we assume that the second argument of `reducing_domain/2` contains the information required for determining the reduced domain. A reduced domain should be neither too small nor too large. A solution is narrowed by this reduction procedure, but it does not normally generate a solution to the problem. Thus, after domain reduction, assignment of values to the variables must be performed, which may also include a search. The main part of the search, however, is carried out by the domain-reducing procedure. A conventional labelling algorithm can be used for the final value assignment. If a contradiction is detected during the final value assignment, the search can backtrack into the reducing procedure.

The domain-reducing strategy is also used in our timetabling system. The advantages of this method have been shown by many examples.

In the domain-reducing strategy, the variable selection and the choice of a reduced domain for the selected variable are nondeterministic. The success of the domain-reducing strategy depends on the chosen heuristics for the order of variable selection and for the determination of the reduced domain. The order of variable selection is determined by the following characteristics of the courses: the given priority of the related subject, and whether the assignment of the starting time or the room has priority. If priority of a subject is not given, then the priority is generated by a random number. The complete order of variable selection is determined by random numbers if necessary. The chosen heuristics for determining the reduced domain take into account wishes with respect to starting times (see constraint C_{11}). Thus, in the first step, it is attempted to reduce the domains of the variables such that the expressed wishes are included in the reduced domain.

Our experience has shown that in many cases either a solution can be found within only a few backtracking steps, or a large number of backtracking steps are needed. We therefore use the following basic search method: the number of backtracking steps is restricted, and different orderings of variable selections are tried out.

- scheduling an individual course
- scheduling marked courses automatically
- removing marked courses from the timetable
- moving an individual course within the timetable
- scheduling the remaining courses automatically

Figure 3 shows a detail of the graphical user interface during scheduling of an individual course. At the bottom of the picture, the time points that can be selected and the preferred starting times (in this case one only) are marked.

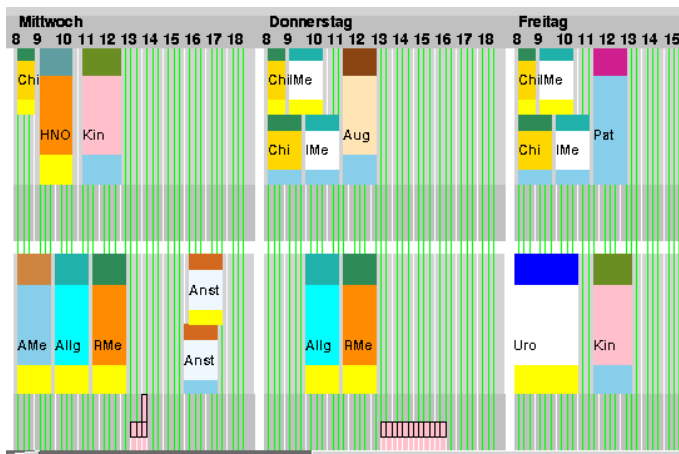


Fig. 3. Detail of the graphical user interface

6 Implementation and Results

The Constraint Logic Programming language CHIP was selected as the implementation language. The global constraints and the object oriented component built into CHIP are particularly useful for problem modelling.

For representation of the problem, we used three phases: definition of the problem, the internal relational representation, and the internal object oriented representation. For the first phase, definition of the problem, we developed a declarative language for problem description. All components of a timetabling problem can be easily defined using this declarative language. Thus, the graphical user interface is only needed to set the parameters. In the second phase, the problem definition is transformed into an internal relational representation. The definition data are converted into a structure that is suitable for finite integer domains. In the third phase, the internal object-oriented representation is generated from the internal relational representation. The object-oriented representation is used for the solution search and the graphical user interface.

The transformation process takes into account the possible cross connections between the definitions and supports the generation of various solutions. The results of the second phase can be used to generate the corresponding objects needed for the different solutions.

Output of the generated timetables is in HTML format. This means that the results of the timetabling are available for further use elsewhere.

In the first phase, the timetables are generated for the clinical-course phase of the studies program (six semesters). Generation of timetables for this course phase had previously caused problems, and involved a great deal of manual work.

The first test phase has been successfully completed. The timetables were generated quickly if there were not any contradictions (violations of hard constraints). Such contradictions were easily detected using the interactive graphical user interface. Conflicts were eliminated in collaboration with the person responsible for timetables at the medical faculty. Knowledge of this field was very important for removing contradictions to ensure that wishes were satisfied in most cases. The results were very well received by all the departments involved.

The generated timetables were output as HTML files and are available in the Internet under <http://www.first.gmd.de/plan/charite>. Figure 4 shows a timetable of lectures from the Internet.

Our timetabling system was given a positive reception by the Charité Medical Faculty at the Humboldt University, Berlin. The timetables were generated quickly and were available students at a very early stage. Allocation of students to course groups was done during the preceding semester, the wishes of the students being taken into account whenever possible.

1. klinisches Semester						Vorlesungsplan
	Montag	Dienstag	Mittwoch	Donnerstag	Freitag	
8 00						
15						
30	Allg. Pathologie	Allg. Pathologie	Pharmakologie			
45	HS Pathologie 8:35 - 9:45	HS Pathologie 8:35 - 9:45	gr. HS Poliklinik 8:35 - 9:45			
9 00				ImmerMed		
15				gr. HS Poliklinik 9:40 - 10:30		
30				Diener Prop. Pathophys.		
45						
10 00	ImmerMed	MikroBio	MikroBio		Radiologie	
15	gr. HS Poliklinik 10:40 - 11:30	Robert-Koch-HS 10:40 - 11:30	Hertwig-HS 10:40 - 11:30		HS 2 10:40 - 11:30	
30	Diener Prop. Pathophys.					
45						
11 00				Patho-BioCh		
15				gr. HS Poliklinik 11:45 - 12:30		
30						
45						
12 00	Patho-BioCh		Immunologie			
15	gr. HS Poliklinik 12:45 - 13:35		Hertwig-HS 12:45 - 13:35			
30						
45						
13 00						
15						
30						
45						
14 00						
Legende: <input checked="" type="checkbox"/> Vorlesung <input type="checkbox"/> Pause						
generiert am 22.12.2007 um 11:40 Uhr						

Fig. 4. A timetable from the Internet

7 Conclusions and Further Work

The initial application of our timetabling system was proved successful and demonstrating the suitability of the methods used. From this application, we were able to obtain useful information for our further work. The importance of a combination of interactive and automatic search was also shown. Further development of our timetabling system will include scheduling courses for any semester, allocation of rooms for all courses, improvements in problem modelling and solution search, and perfection of the graphical user interface. Research is needed to complete our interactive, automated timetabling system for the medical faculty. Studying of the influence of different modelling techniques on the solution search and exploring techniques for heuristic solution search will be the main elements of this research. The methods, techniques and concepts developed or under development will also be tested on other applications.

References

1. S. Abdennadher and M. Marte. University timetabling using constraint handling rules. In O. Ridoux, editor, *Proc. JFPLC'98, Journées Francophones de Programmation Logique et Programmation par Contraintes*, pages 39–49, Paris, 1998. Hermes.
2. A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *J. Mathematical and Computer Modelling*, 17(7):57–73, 1993.
3. F. Azevedo and P. Barahona. Timetabling in constraint logic programming. In *Proc. World Congress on Expert Systems*, 1994.

4. E. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *J. Mathematical and Computer Modelling*, 20(12):97–123, 1994.
5. P. Boizumault, Y. Delon, and L. Peridy. Constraint logic programming for examination timetabling. *J. Logic Programming*, 26(2):217–233, 1996.
6. E. Burke and M. Carter, editors. *Practice and Theory of Automated Timetabling II*, volume 1408 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, 1998.
7. E. Burke, D. Eililman, P. Ford, and R. Weare. Examination timetabling in British universities: A survey. In [8], pages 76–90, 1996.
8. E. Burke and P. Ross, editors. *Practice and Theory of Automated Timetabling*, volume 1153 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, 1996.
9. T. B. Cooper and J. H. Kingston. The complexity of timetable construction problems. In [8], pages 183–295, 1996.
10. M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Int. Conf. Fifth Generation Computer Systems (FGCS'88)*, pages 693–702, Tokyo, 1988.
11. T. Frühwirth. Theory and practice of constraint handling rules. *J. Logic Programming*, 37:95–138, 1998.
12. H.-J. Goltz. Reducing domains for search in CLP(FD) and its application to job-shop scheduling. In U. Montanari and F. Rossi, editors, *Principles and Practice of Constraint Programming – CP'95*, volume 976 of *Lecture Notes in Computer Science*, pages 549–562, Berlin, Heidelberg, 1995. Springer-Verlag.
13. C. Guéret, N. Jussien, P. Boizumault, and C. Prins. Building university timetables using constraint logic programming. In [8], pages 130–145, 1996.
14. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge (Mass.), London, 1989.
15. M. Henz and J. Würtz. Using Oz for college timetabling. In [8], pages 162–177, 1996.
16. M. Kambi and D. Gilbert. Timetabling in constraint logic programming. In *Proc. 9th Symp. on Industrial Applications of PROLOG (INAP'96)*, Tokyo, Japan, 1996.
17. G. Lajos. Complete university modular timetabling using constraint logic programming. In [8], pages 146–161, 1996.
18. K. Marriott and P. J. Stucky. *Programming with Constraints: An Introduction*. The MIT Press, Cambridge (MA), London, 1998.
19. A. Schaerf. A survey of automated timetabling. Technical Report CS-R9567, Centrum voor Wiskunde en Informatica, 1995.
20. G.M. White and J. Zhang. Generating complete university timetables by combining tabu search with constraint logic. In [9], pages 187–198, 1998.

Constraint-Based Resource Allocation and Scheduling in Steel Manufacturing

Mats Carlsson, Per Kreuger, and Emil Åström

Intelligent Systems Laboratory
Swedish Institute of Computer Science
Box 1263, SE-164 29 Kista, Sweden
{matsc,piak,emil}@sics.se
Phone: +46-8-633 1500, Fax: +46-8-751 7230

Abstract. This paper describes a flow maximization problem in steel manufacturing, decomposes it into three sub-problems, and models them in terms of finite domain constraints. Greedy algorithms are used for solving the sub-problems. The constraints are used for maintaining consistency rules, not for optimization. A tool implementing these algorithms and equipped with a GUI has been implemented.

Keywords: flow maximization, scheduling, finite domain constraints, steel manufacturing.

1 Introduction

Ovako Steel is the leading European producer of bearing steel and a prominent manufacturer of special engineering steel. Ovako Steel belongs to the SKF group and most of the steel production is done in Hofors, Sweden. This facility is organized as in Fig. 1. The business of the facility is to process scrap and pure metals into steel of various dimensions and grades, satisfying shop orders.

In the process, the steel is handled at different levels of granularity:

- A *batch* is the largest entity, about 100 tons. The steel arrives from the melting shop to the rolling mill in batch sizes and the entire batch is charged into a single pit furnace.
- The *batch queue* is the predicted sequence of batches to arrive from the steel works and their properties and arrival times.
- A batch is normally an aggregate of 24 *ingots*. An ingot weighs about 4.2 tons. The steel is decharged from the pit furnaces one ingot a time.
- A *billet* is the final product of the rolling mill. Several billets are made from one ingot by cutting it and rolling it. Billets have several different lengths and dimensions and a square or round *shape* (cross-section).
- A *suite* consists of all ingots of a batch that are assigned to the same shop order. All billets from an ingot are assigned to the same shop order.

A *shop order* specifies a steel grade, a billet length, amount, dimension and shape, an end temperature, a minimal soaking time, and a due date.

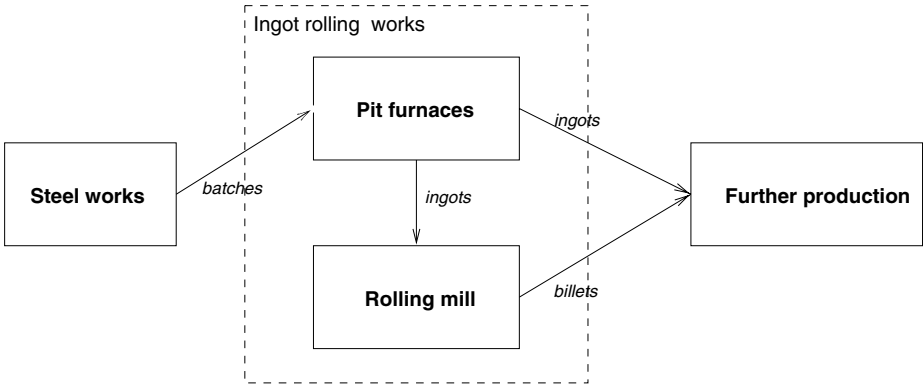


Fig. 1. Overview of the Hofors facility

Scrap metal is melted and cast in the steel works, producing solid ingots at a temperature of about 900°C , which are transported to the ingot rolling works. Here, the ingots are optionally submerged in water, then heated in the pit furnaces to about 1200°C , then soaked at that temperature for several hours, and finally rolled and cut into billets, which is done in the rolling mill. The billets are then stored for further production. Some ingots are only heated and soaked and not processed into billets.

The objective is to maximize the throughput of the ingot rolling works. The twelve pit furnaces (see Fig. 2) are central to the scheduling since the flow is highly dynamic and the furnaces serve as a kind of buffer. A detailed schedule spanning over several days is meaningless, since there are different levels of uncertainty at every step of the production, both in terms of timing and in the physical result of the operations (the quality of the products for example). Instead, a schedule for the current contents of the pit furnaces and the batch queue is computed whenever some event occurs that falsifies some assumption in the current schedule. Many potential problems can be avoided by making the correct decisions about the furnaces.

1.1 The Ingots Rolling Works

The ingot rolling works has two main production units: the pit furnaces and the rolling mill, as shown in Fig. 3. The ingots are *charged* into the furnaces for heating and soaking. They are then *decharged* from the furnaces and rolled in a 220mm roller. They are then dispatched to one of 3 flows depending on the shape and dimension of the finished product. In these flows, operations such as rolling and cutting are applied to the steel. Two of the flows merge at a late stage.

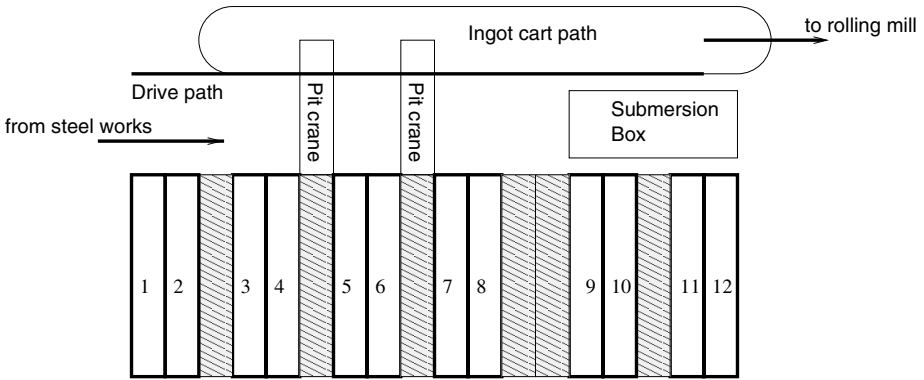


Fig. 2. The pit furnaces

The initial 220mm roller has the capacity to process one ingot every three minutes. However, some material must be processed at a slower pace to avoid stalls further downstream. Three minutes is the *cycle time* for the 220mm roller.

There are many constraints at the different steps of the production. The pit furnaces have individual properties, so the time required for heating differs between them. Heating takes longer if the steel is delayed before it is charged, as it is then cooler when charged into the furnace. The pit cranes, used for charging and decharging, must not come too close to each other.

Furthermore, the cutting operation has a fixed capacity and so the number of cuts per ingot affects the maximum decharge pace. The medium round flow handles many different dimensions and often needs reconfiguration, which takes time. Two suites must not be interleaved in any flow. However, there are significant opportunities for processing the suites simultaneously in the parallel flows.

1.2 Outline

The rest of the paper is organized as follows. In Section 2 we describe how the problem is decomposed into two resource allocation problems and a scheduling problem, and modeled with finite domain constraints. Certain simplifications and abstractions are also described. In Section 3, a greedy algorithms for solving the first resource allocation problem, and another one for solving the remaining subproblems, are given. Experiences from using a global search approach are also given. We have implemented these algorithms as a scheduling and decision support tool. This tool and its integration into the operating environment is described in Section 4. We end with some conclusions.

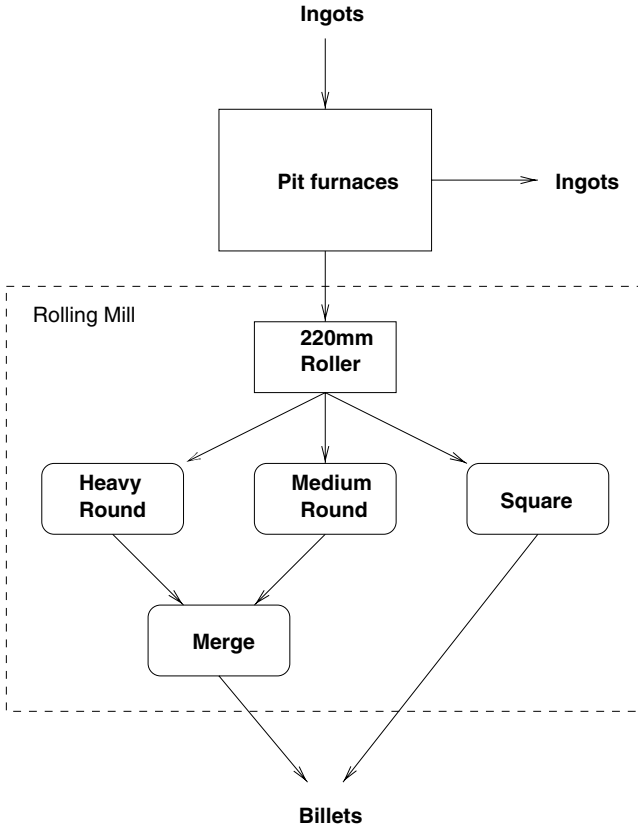


Fig. 3. Flow model: from ingots to billets

2 Modeling the Problem

Some simplifications and abstractions have been made. For example, the two round flows are modeled as a single flow. This is possible since the two round flows are hardly ever used in parallel (confirmed both by users and historical data). This abstraction makes it possible to disregard the complication of the two merging flows. The crane movements are not modeled explicitly. Instead, we impose constraints on which combinations of furnaces can be charged and discharged simultaneously.

The constraint-solver implements the constraint model and applies heuristic search to create schedules. Global search for optimal solutions is not done because of difficulties in stating a valid cost measure and computational complexity. The solver is implemented using the CLPFD (finite domain constraints) library of SICStus Prolog [4] and only local constraints are currently being used.

In the rest of this section, we decompose the problem and itemize the parameters of a problem instance, which are functions of the batch queue and the shop order assignment. We then state the problem variables: furnaces, charge times, and decharge times. Finally, we state the objective function and problem constraints.

2.1 Problem Structure

Three sub-problems have been identified: two resource allocation problems (1-2) and one scheduling problem (3):

1. Selecting shop orders and assigning ingots to them. The way this is done greatly affects the scheduling since the dimensions, heating and soaking requirements, etc., are important factors for how difficult the scheduling will be. This sub-problem is solved first and is not part of the constraint model. Its solution instead determines a number of parameters of the latter. Its algorithm is described in Section 3.1.
2. Assigning batches to furnaces. This is also important because of the different properties of the different furnaces and the crane movement constraints.
3. Scheduling the ingot operations, i.e. deciding when to charge and decharge the ingots and in what order. This problem is roughly a 2-machine job-shop scheduling problem [1, 3] with sequence-dependent setup times [2], due to reconfigurations. However, there are complex interactions between the jobs: the crane movement constraints and the non-interleaving suites constraints. The latter force the scheduler to reason at both the suite and the ingot levels. At the suite level, durations are variable. These complications mean that the problem cannot be readily reduced to a standard scheduling problem.

2.2 Problem Parameters

Here, $j, j' \geq 1$ denote suite numbers, $b \geq 1$ denotes a batch number, and $1 \leq f \leq 12$ denotes a furnace number.

$P_1(j)$, $P_2(j)$ The maximal decharge pace for suite j expressed as “at most $P_1(j)$ out of $P_2(j)$ cycles in the 220mm roller”.

For square billets and round billets ≥ 160 mm, $P_1(j) = P_2(j) = 1$, i.e. the ingots can be decharged and rolled at full pace. Table 1 relates dimensions and number of cuts to $(P_1(j), P_2(j))$ for other dimensions.

SETUP(j, j') Setup time for suite j' if its predecessor is suite j . Only relevant for the round flow, and only for dimensions ≤ 160 mm. The setup time varies between 0 and 20 minutes, depending on the dimensions of j and j' .

In particular, the setup time is zero if they are of the same dimension.

NJ Number of suites.

NB Number of batches.

Dim/div	≤ 5	6	7	8	9	10	≥ 11
78	(1,2)	(1,2)	(1,2)	(1,2)	(1,2)	(1,2)	(1,3)
80	(1,2)	(1,2)	(1,2)	(1,2)	(1,2)	(1,2)	(1,3)
85	(1,2)	(1,2)	(1,2)	(1,2)	(1,2)	(1,2)	(1,3)
90	(2,3)	(2,3)	(1,2)	(1,2)	(1,2)	(1,2)	(1,3)
95	(2,3)	(2,3)	(1,2)	(1,2)	(1,2)	(1,2)	(1,3)
100	(2,3)	(2,3)	(1,2)	(1,2)	(1,2)	(1,2)	(1,3)
105	(2,3)	(2,3)	(1,2)	(1,2)	(1,2)	(1,2)	(1,2)
110	(1,1)	(2,3)	(2,3)	(2,3)	(1,2)	(1,2)	(1,2)
115	(1,1)	(2,3)	(2,3)	(2,3)	(2,3)	(2,3)	(2,3)
120	(1,1)	(2,3)	(2,3)	(2,3)	(2,3)	(2,3)	(2,3)
125	(1,1)	(2,3)	(2,3)	(2,3)	(2,3)	(2,3)	(2,3)
130	(1,1)	(2,3)	(2,3)	(2,3)	(2,3)	(2,3)	(2,3)
135	(1,1)	(2,3)	(2,3)	(2,3)	(2,3)	(2,3)	(2,3)
140	(1,1)	(2,3)	(2,3)	(2,3)	(2,3)	(2,3)	(2,3)
150	(1,1)	(2,3)	(2,3)	(2,3)	(2,3)	(2,3)	(2,3)

Table 1. (P_1, P_2) for round dimensions

$\text{CMIN}(b), \text{CMAX}(b), \text{CDUR}(b)$ Earliest and latest charge time and duration for batch b . The charge duration $\text{CDUR}(b)$ is in the order of 1-2 minutes per ingot. $\text{CDUR}(b)$ depends on whether the ingots need to be submerged into water before charging.

$\text{SUITES}(b)$ The set of suites of batch b .

$\text{SHAPE}(j)$ The shape of suite j (round or square).

$\text{I}(j)$ The set of ingots of suite j . $\text{I}(j) = j_0..j_n$ is a consecutive set of integers, imposing a decharge order within the suite.

$\text{HDUR}(j, f)$ The heating duration for suite j and furnace f , if charged on time.

$\text{SDUR}(j)$ The soaking duration for suite j .

2.3 Problem Variables

$F(b) \in 1..12$ Furnace number of batch b .

$T_c(b) \in -\infty..\infty$ Charge time for batch b . $T_c(b) < 0$ if the batch has already been charged at the beginning of the period to be scheduled.

$T_d(i) \in 0..\infty$ Decharge time for ingot i .

2.4 Objective Function

Traditionally, scheduling research has concentrated its efforts on developing algorithms for makespan minimization. In our application, however, the objective is to maximize a perpetual flow, rather than to minimize the makespan of a fixed set of tasks. Furnaces tend to be bottlenecks, so we minimize their total busy time:

$$\begin{aligned}
 \text{Cost} &= \sum_{b \in 1..NB} (\max\{T_d(j_n) \mid \exists j : j \in \text{SUITES}(b)\} - T_c(b)) \\
 \text{where} \\
 I(j) &= j_0..j_n
 \end{aligned} \tag{1}$$

The objective function is only used implicitly—our greedy algorithms have heuristics designed to minimize it.

2.5 Problem Constraints

Charge Time constraints Each charge time is bounded from below by its forecasted arrival time. If allowed to cool too long, the steel can't be used in normal production; hence an upper bound on the charge time exists. Finally, overtaking is disallowed: batches must be charged in arrival order.

$$\begin{aligned}
 \forall b \in 1..NB : \\
 \text{CMIN}(b) &\leq T_c(b) \leq \text{CMAX}(b)
 \end{aligned} \tag{2}$$

$$\begin{aligned}
 \forall b \in 1..NB - 1 : \\
 T_c(b) + \text{CDUR}(b) &\leq T_c(b + 1)
 \end{aligned} \tag{3}$$

Decharge Order Constraints These constraints serve two purposes. Firstly, they ensure that the dimension-dependent capacity of the respective flow is not exceeded. Secondly, they break symmetries by imposing a decharge order within each suite. Constraint 4 ensures that a prescribed minimal time elapses between two consecutive dechargings. Constraint 5 is needed for the case where at most two out of three cycles may be used in the 220mm roller.

$$\begin{aligned}
 \forall j \in 1..NJ \\
 T_d(j_0) + \delta_1(j) &\leq \dots \leq T_d(j_n)
 \end{aligned} \tag{4}$$

$$\begin{aligned}
 \forall j \in 1..NJ \\
 T_d(j_0) + \delta_2(j) &\leq T_d(j_2) + \delta_2(j) \leq \dots \leq T_d(j_{n'}) \\
 \forall j \in 1..NJ \\
 T_d(j_1) + \delta_2(j) &\leq T_d(j_3) + \delta_2(j) \leq \dots \leq T_d(j_{n''})
 \end{aligned} \tag{5}$$

where $j_0..j_n$ denotes the set $I(j)$, (n', n'') denotes $(n, n - 1)$ if n is even and $(n - 1, n)$ otherwise, and

$$\delta_1(j) = 3 \times \begin{cases} P_2(j), & \text{if } P_1(j) = 1 \\ 1, & \text{otherwise} \end{cases}, \quad \delta_2(j) = 3 \times \begin{cases} 2P_2(j), & \text{if } P_1(j) = 1 \\ P_2(j), & \text{if } P_1(j) = 2 \\ 2, & \text{otherwise} \end{cases}$$

Heating and Soaking Time Constraints These are precedence constraints between charge and decharge times for each batch and suite. They also encode the rule of thumb that delaying a charging by n minutes increases the heating duration by $2n$ minutes.

$$\begin{aligned}
& \forall b \in 1..NB, j \in \text{SUITES}(b) \\
& 3 \times T_c(b) - 2 \times \text{CMIN}(b) + \text{CDUR}(b) + \text{HDUR}(j, F(b)) + \text{SDUR}(j) \leq T_d(j_0) \\
& \text{where} \\
& I(j) = j_0..j_n
\end{aligned} \tag{6}$$

Exclusive Use of Furnaces These constraints model the fact that a furnace must be emptied before a new batch can be charged into it.

$$\begin{aligned}
& \forall b, b' \in 1..NB \mid b < b' \wedge F(b) = F(b') : \\
& \quad \forall j \in \text{SUITES}(b) : T_d(j_n) \leq T_c(b') \\
& \text{where} \\
& I(j) = j_0..j_n
\end{aligned} \tag{7}$$

Non-shared Common Resource These constraints model the fact that the 220mm roller has a limited capacity and needs to process every ingot. They only need to consider pairs of ingots of different shapes. The case with ingots of the same shape is covered by constraints [4] and [9]

$$\begin{aligned}
& \forall i, j \in 1..NJ \mid i < j \wedge \text{SHAPE}(i) \neq \text{SHAPE}(j) : \\
& \quad \forall k \in 0..n, l \in 0..n' : \\
& \quad \quad T_d(i_k) + 3 \leq T_d(j_l) \vee \\
& \quad \quad T_d(j_l) + 3 \leq T_d(i_k) \\
& \text{where} \\
& I(i) = i_0..i_n, I(j) = j_0..j_{n'}
\end{aligned} \tag{8}$$

Non-interleaving Decharging of Suites These constraints model the fact that a suite must not be interleaved with any other suite of the same shape.

$$\begin{aligned}
& \forall i, j \in 1..NJ \mid i < j \wedge \text{SHAPE}(i) = \text{SHAPE}(j) : \\
& \quad T_d(i_n) + \delta_1(i) + \text{SETUP}(i, j) \leq T_d(j_0) \vee \\
& \quad T_d(j_{n'}) + \delta_1(j) + \text{SETUP}(j, i) \leq T_d(i_0) \\
& \text{where} \\
& I(i) = i_0..i_n, I(j) = j_0..j_{n'}
\end{aligned} \tag{9}$$

where δ_1 is defined in [4].

Charge-Decharge Constraints These constraints express the fact that overlapped charging and decharging is allowed only if the furnaces are not too close. These constraints capture the crane movements constraints.

$$\begin{aligned}
& \forall b, b' \in 1..NB \mid b \neq b' \wedge \neg C(F(b), F(b')) : \\
& \quad \forall j \in \text{SUITES}(b') : \\
& \quad \quad T_c(b) + \text{CDUR}(b) \leq T_d(j_0) \vee \\
& \quad \quad T_d(j_n) \leq T_c(b) \\
& \text{where} \\
& I(j) = j_0..j_n
\end{aligned} \tag{10}$$

where $C(f, f')$ is true iff charging furnace f may overlap with decharging furnace f' .

3 Algorithms

Initially, we used a monolithic global search approach to the full problem, with makespan as the objective function to minimize. This approach showed extremely poor convergence properties, and was abandoned early on, particularly as it was realized that makespan minimization is not relevant to our problem.

We then decomposed the problem into the three subproblems, using a greedy algorithm for each sub-problem, and solved them in sequence. However, this approach turned out to be too suboptimal—the furnace assignment algorithm produced assignments which led to poor schedules later.

Having realized that furnace assignment needs to be intertwined with ingot scheduling, we finally decomposed the problem into two phases which are solved in sequence. The first phase corresponds to the shop order assignment problem. It tries to find an assignment that could potentially produce a good schedule, while respecting due dates of shop orders. The second phase is responsible for furnace assignment and ingot scheduling. It has built-in heuristics designed to minimize the objective function (□). Greedy algorithms are used in both phases.

3.1 Shop Order Assignment

The purpose of this algorithm is to select shop orders and to assign ingots to them. In terms of the constraint model, this amounts to fixing the suites and all suite related problem parameters. The other parameters are determined by the batch queue alone. The algorithm has two phases: a first-fit phase, and a balancing phase.

The First-Fit Phase For each $b \in 1..NB$, in increasing order:

We search the shop orders, by increasing due date, until a shop order o for a steel grade matching that of b is found. As many ingots as possible from b are assigned to o . If there are any ingots from b left, we keep searching for another matching shop order. If there is no matching shop order, any residual ingots will not be assigned.

The Balancing Phase The purpose of the balancing phase is to replace some of the assignments of shop orders for round material into shop orders for square material, in order to increase the throughput, mainly by increasing the amount of available parallelism between the flows.

Table□ defines cycle times for medium round dimensions. For example, ingots to be cut 5 times and rolled to 100mm billets can only be decharged two out of three cycles; otherwise, the round flow would stall. Thus to keep the 220mm

roller busy, the third cycle should ideally be filled by an ingot destined for the square flow.

This phase tries to reassign shop orders so that the number of idle cycles is minimized, assuming an ideal schedule:

While there are still idle cycles, the algorithm searches for a shop order assignment that can be changed so that a maximal number of idle cycles are filled.

The search prioritizes assignments with late due dates. Overdue shop orders are not reassigned.

3.2 Furnace Assignment and Ingot Scheduling

In this context, a *task* is either the charging of a batch into a furnace, or the discharging of a suite from a furnace. The general idea is to obtain a complete schedule by incrementally inserting tasks into an initially empty schedule. The algorithm is analogous to a simulation of the time period being scheduled. A major complication is the charge-decharge constraint (II). To maximize throughput, a charge task should always run in parallel with a decharge task if possible.

We first try to schedule a decharge task for a *ready* suite that can be completed before the earliest start time of any charge task. A suite is ready if it has been fully heated and soaked. If no such decharge task is available, the next charge task is scheduled, heuristically choosing a furnace known to be empty at that time. This is iterated until all tasks have been scheduled.

A suite is not considered for discharging if there is another suite that could be discharged earlier. This is the key heuristic for minimizing setup times—the earliest decharge time of a ready suite requiring no roller reconfiguration is smaller than that of a ready suite requiring reconfiguration due to a dimension change.

A square suite s is not considered for discharging if there is a round suite that could be discharged before s has been completely discharged. The point is that square and round suites can be discharged in parallel, and that square suites can be used to exploit idle capacity when the 220mm roller is processing a round suite at less than full pace. This is best done by scheduling round suites first and filling any idle roller cycles with square ingots later.

A suite is not considered for discharging if that could delay the next charge task due to constraint (II). A delayed charge task entails a longer heating duration, which could lead to cascaded delays. Conversely, scheduling a charge task more eagerly might impose too tight constraints on the decharge tasks.

If a ready suite can be scheduled for discharging, the task with the earliest possible start time is chosen, breaking ties according to a heuristic, selected by the user. The heuristics implemented to date are (a) preferring the suite that has waited the longest, and (b) choosing a suite that will help empty the furnace it occupies as soon as possible. Both heuristics are designed to minimize the objective function (II).

If a charge task must be scheduled, the algorithm selects the furnace that minimizes the number of neighbor furnaces with ready suites that would be blocked by constraint [10](#). Ties are broken by selecting the most powerful furnace, thus minimizing the heating duration, and contributing to minimize the objective function.

In the background, constraints [\(8,9,10\)](#) are actively pruning the earliest and latest start and completion times of the tasks, which constitutes the key input to the algorithm. The algorithm furthermore maintains the following state variables, which are initialized to correspond to the situation at the start of the schedule:

- The batch queue Q , which imposes an order on the charge tasks (constraint [3](#)).
- For each furnace f : an ingot count, the completion time of the most recent task involving f , and the set of suites being heated in f .

The algorithm is shown below. Let $\text{est}(s)$ ($\text{ect}(s)$) denote the earliest start (completion) time of discharging suite s . Let R (S) denote the set of ready, round (square) suites. Time variables are fixed by assigning them to their earliest possible values:

decharge:

```

 $E_c \leftarrow$  the earliest charge time of  $\text{head}(Q)$ , or  $\infty$  if  $Q = \emptyset$ ;
 $E_r \leftarrow \min\{\text{est}(s) \mid s \in R\}$ ;
 $E_s \leftarrow \min\{\text{est}(s) \mid s \in S\}$ ;
 $C_r \leftarrow \{s \mid s \in R \wedge \text{est}(s) = E_r \wedge \text{ect}(s) \leq E_c\}$ ;
 $C_s \leftarrow \{s \mid s \in S \wedge \text{est}(s) = E_s \wedge \text{ect}(s) \leq \min(E_c, E_r)\}$ ;
if ( $C_s \neq \emptyset$ ) select  $s \in C_s$  breaking ties as described above;
else if ( $C_r \neq \emptyset$ ) select  $s \in C_r$  breaking ties as described above;
else goto charge;
fix the decharge times of all ingots of  $s$  subject to constraints 4, 5;
update the state and goto decharge;
```

charge:

```

if ( $Q = \emptyset$ ) exit;
select the furnace for  $\text{head}(Q)$  according to the above heuristics;
fix the charge time for  $\text{head}(Q)$ ;
 $Q \leftarrow \text{tail}(Q)$ ;
update the state and goto decharge;
```

4 The Application

The operating environment is shown in Fig. [4](#).

The input to the tool is in the form of a snapshot file. This file contains all necessary information for the scheduling. Via the tool, the user can:

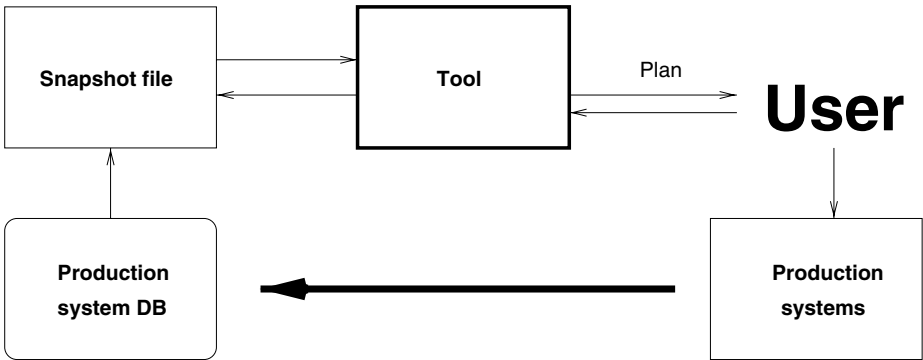


Fig. 4. Operating environment

- initiate snapshot file generation and import a snapshot file,
- perform semi-automatic shop order selection and assignment
- compute, inspect, store and compare schedules
- edit and add parameters such as planned stops

Thus, the user can use the tool to compute schedules for multiple scenarios, until a satisfactory schedule is obtained.

The output of the tool is currently not directly connected to the actual production systems at Ovako Steel. Rather, the user is given a suggestion for a schedule which (s)he then can base the operational decisions on (i.e. control the production systems). A new schedule can be generated at any time by loading a new snapshot file since the snapshot file is always based on up-to-date data from the production systems.

The GUI plays a central role in the application. It reflects the breakdown into sub-problems (one problem is solved separately from the two others) in that there is one application window for assigning ingots to shop orders and one window where the assignment of batches to furnaces and the scheduling is controlled. The GUI was written in Tcl/Tk, using the Tcl/Tk interface of SICStus Prolog.

The first window (the *allocation window*) allows the user to do shop order allocation semi-automatically. The idea is that the user does the important parts of the allocation manually and then applies the automatic procedures to complete it. This way, the user has full control of the process while avoiding unnecessary work. Another way to use this part of the application is to let the automatic functions create an initial allocation and then modify this by hand.

The second window (the *main window*) gives the user the opportunity to initiate snapshot file and schedule generation and inspect the generated schedules displayed as Gantt charts. Other functionality includes the possibility to set parameters for the scheduling, add planned stops for the furnaces and the roller flows, load and save entire situations for later discussions or debugging sessions, and to store and compare schedules.

A screen shot of the main window is shown in Fig. 5. The histogram near the bottom indicates the number of furnaces containing steel that is ready for discharging. Ideally, the number should not reach zero, and generally should not vary too much. The three bars above the histogram show the activity in the various flows. In this example, there is little parallelism. The medium round flow is interrupted several times due to roller reconfigurations (dimension changes), during which time the square flow is active. Finally, each pit furnace is displayed as a bar indicating its various processing phases.

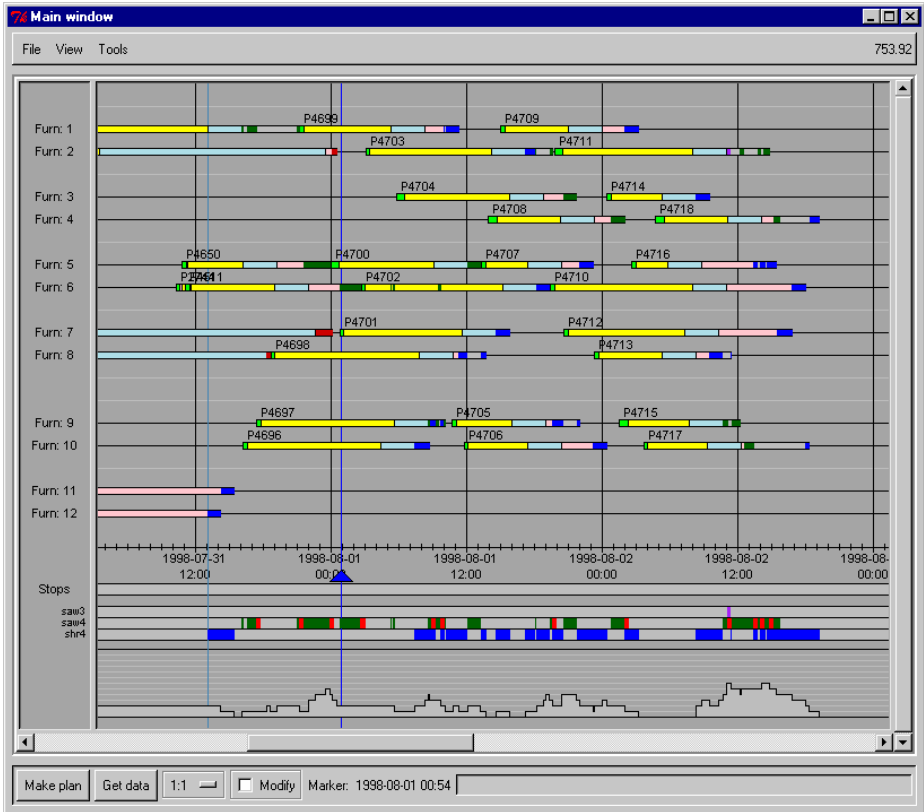


Fig. 5. A schedule displayed as a Gantt chart

In general, user integration has been an important element in the development of the application. Prototypes were developed early in the project and the GUI has gradually evolved into its current state. The active role of the user is evident both in the allocation window and in the main window. In the latter there are for example functions for comparing schedules; it is the user that selects the best schedule, not the application. There are also many small functions

for making it easy to inspect the schedules. It is always the user who has the initiative; the application does not do anything except as reactions to user actions. The generated schedules are used as advice for the user, not as rules, since it is always the user that controls the production systems.

5 Conclusions

The work reported herein is still in progress. Hence, a full evaluation of the algorithms has not yet been made.

The problem described herein exhibits many properties that we believe are typical for constraint programming approaches to real-world problems:

- The problem is a combination of several sub-problems with many complex side-constraints, and hence cannot be reduced to a standard problem.
- Soft constraints are avoided by construction, as they are computationally inefficient in constraint programming.
- A strict objective function that makes sense and is computationally efficient has been difficult to identify. A complex cost function renders global search infeasible, due to poor constraint propagation. Instead, decomposing the problem into sub-problems and solving these sequentially using greedy algorithms led to vast improvements in efficiency.
- It is necessary to strike a balance between optimality and simplicity. A minor sacrifice in optimality can lead to major simplifications and performance gains.
- Constraints are excellent for maintaining consistency rules in an algorithmic setting. Using constraints doesn't imply global search.

CLP proved to be not only adequate for the problem, but crucial in delivering the rapid prototyping and easy modification that was necessary in order to complete the inevitable development cycles within the tight project deadlines.

All prototypes were equipped with a GUI. Early design of GUIs helps in knowledge acquisition and in identification of functional needs. The GUI was also instrumental in pinpointing deficiencies in the schedules generated by the prototypes, and was crucial for the end users' acceptance of the scheduling tool.

The application required a high degree of user integration and CLP proved to be capable in that area. This confirms the observations in [5].

Acknowledgements

This work was carried out within ESPRIT project TACIT¹ (Trial Application using Constraint programming in Industrial manufacTuring). Special thanks to Ovako Steel for providing the crucial domain knowledge and application data. Péter Szeredi's helpful comments are gratefully acknowledged.

¹ URL: <http://www.sics.se/col/projects/tacit/>

References

- [1] K.R. Baker. *Introduction to sequencing and scheduling*. Wiley & Sons, 1974.
- [2] P. Bruckner and O. Thiele. A branch & bound method for the general-shop problem with sequencing dependent setup-times. *OR Spectrum*, 18:145–161, 1996.
- [3] J. Carlier and E. Pinson. An algorithm for solving the job-shop scheduling problem. *Management Science*, 35(2):164–176, 1989.
- [4] Mats Carlsson, Greger Ottosson, and Björn Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. Hartel, and H. Kucklen, editors, *PLILP'97, International Symposium on Programming Languages: Implementations, Logics, and Programming*, number 1292 in Lecture Notes in Computer Science, Southampton, September 1997. Springer-Verlag.
- [5] André Chamard, Annie Fischler, Dominique-Benoît Guinaudeau, and André Guillaud. CHIC lessons on CLP methodology. Technical Report ESPRIT EP5291 / D2.1.2.3, CHIC Project / Dassault Aviation / Bull, January 1995. (Constraint Handling in Industry and Commerce).

Using Constraints in Local Proofs for CLP Debugging

Claude Lai

Société PrologIA
Case 909, 163 Avenue de Luminy
F-13288 Marseille Cedex 9 FRANCE
`lai@prologianet.univ-mrs.fr`

Abstract. This paper describes the addition of assertions for CLP debugging. These assertions are formed by a set of preconditions and a set of postconditions for the program predicates. We present a system with a rich set of primitives (based on constraints and meta-properties) to express the program properties. These primitives are such that they can often be proved efficiently with a local analysis at compile-time. In case of failure of the proof procedure, the program is completed with run-time checks to insure a correct execution.

1 Introduction

The frame of this work is the DiSCiPl Project [\[1\]](#). The goal of this project is to define, to implement and to assess novel and effective debugging systems for Constraint Programming. It is considered there are two important aspects to help the debugging of CP applications: first, the choice of a programming methodology, and second, the selection of a validation procedure. Validation is the verification of the correspondence between the program and the programmer's intention. It can be modeled by the notions of partial correctness of programs and completeness of solutions. The main problem is to locate bugs when symptoms of incorrectness or incompleteness have been observed. Since CP is a declarative paradigm, it is particularly well suited to apply declarative analysis methods such as assertion based methods used in validation [\[5\]](#), and declarative debugging techniques used to locate bugs [\[1,8,7\]](#).

As a matter of fact, the introduction of assertions in programs addresses both of these aspects. Assertions are partial specifications which are provided by the programmer. They can be attached to the program predicates and they represent expected meaning of the predicates. Assertions are associated to the notion of correctness. A special kind of assertions, concerning the execution of a program with the Prolog computation rule, has been discussed in [\[6\]](#). The described properties are sometimes called dynamic or run-time properties. A

¹ Debugging Systems for Constraint Programming, Long Term Research Project, Esprit Proposal Number 22532, EURL: <http://discipl.inria.fr>

special kind of run-time assertions, where the verification conditions becomes simpler, is known under the name of directional types [3].

The recent work of [2] describes a refinement of the verification conditions for directional types, and [9] shows that these types are applicable for the debugging of CP programs, where the assertions are considered as constraints on an extended constraint structure, their proof being performed using abstract interpretation techniques. This paper describes a continuation of this work based on the same ideas. The practical aspect of this work is obviously to give a powerful debugging tool to the user. This tool allows to introduce a good specification of programs and performs automatic verifications of these programs.

In this article, we will be referring to Prolog with constraints as the language used to write the programs we want to prove. Prolog IV [8] is the support used for the examples of constraints.

This paper is organized as follows: first, the form of the assertions is defined; then a verification of the program at compile time is described; finally a transformation of the program is presented which allows a fair execution of rules which assertions could not be proved.

2 Form of Assertions

2.1 Syntax and Meaning of Assertions

Assertions are expressed with a new directive handled by a compiler preprocessor. For a given predicate, an assertion is formed by a set of preconditions and a set of postconditions. It is described in the following syntax:

$:- \text{pred}(X_1, X_2, \dots, X_n) : (\text{pre}_1, \text{pre}_2, \dots, \text{pre}_p) \Rightarrow (\text{post}_1, \text{post}_2, \dots, \text{post}_q).$
where:

- *pred* is the name of the predicate;
- X_1, X_2, \dots, X_n are distinct variables representing the arguments of the predicate;
- $\text{pre}_1, \text{pre}_2, \dots, \text{pre}_p$ represent the set of preconditions;
- $\text{post}_1, \text{post}_2, \dots, \text{post}_q$ represent the set of postconditions.

The interpretation of such an assertion is:

- the set of preconditions is interpreted as a conjunction and defines the needed properties of an admissible model for the given definition of the predicate *pred*;
- the set of postconditions is interpreted as a conjunction of the properties of the set of solutions satisfying the predicate *pred* in that model.

In an operational view, assertions have a meaning very similar to the procedure contracts in the Eiffel language: the caller must guarantee the truth of the preconditions before the predicate call, and the predicate definition must guarantee that the postconditions will hold after execution.

² Prolog IV is an operational product developed and marketed by PrologIA, Parc scientifique de Luminy, case 919, 13288 Marseille Cedex 9, France. URL: <http://prologianet.univ-mrs.fr>

2.2 Conditions Allowed in Assertions

We have now to define the expected form of the conditions in the assertions. In our case, it is possible to mix two different kinds of conditions: predefined constraints of Prolog IV solvers which admit an opposite constraint, and more general "meta" conditions which we will handle with abstract domains. Prolog IV has two main solvers for operations on numbers: a linear solver which is complete, and an interval solver (handling a very rich set of linear and non linear operations) which is not. However, there is, for the interval solver, an extension of the usual structure, for which this solver is complete. It is such that any solution in the real structure is also a solution in the extended structure and, if there is no solution in the extended structure, there is also no solution in the real structure. Therefore, a classical proof by inconsistency with the negation of a property is sound with this solver too: if there is no solution in the extended structure it is guaranteed there is none in the real one.

Predefined Constraints

- Domain constraints:
 - $int(X)$ constrains X to only have integer solutions;
 - $nint(X)$ constrains X not to have integer solutions;
 - $real(X)$ constrains X to only have number solutions;
 - $nreal(X)$ constrains X not to have number solutions;
 - $list(L)$ constrains L to only have list solutions;
 - $nlist(L)$ constrains L not to have list solutions;
- Interval constraints:
 - $cc(X, B_1, B_2)$ which constrains X solutions to belong to the interval $[B_1, B_2]$;
 - $outcc(X, B_1, B_2)$ which constrains X solutions to be out of the interval $[B_1, B_2]$;
 - $co(X, B_1, B_2)$ which constrains X solutions to belong to the interval $[B_1, B_2]$;
 - $outco(X, B_1, B_2)$ which constrains X solutions to be out of the interval $[B_1, B_2]$;
 - $Z = K . * . Y$, with K a constant number. This relation is handled by the interval solver. It establishes a relationship between two numbers X and Y and their product Z ;
 - $Z = X . + . Y$. This relation is handled by the interval solver. It establishes a relationship between two numbers X and Y and their sum Z ;
 - $X \text{ le } Y$ ("less or equal" handled by the interval solver);
 - $X \text{ gt } Y$ ("greater than" handled by the interval solver);
 - other similar comparison operators ...
- Linear constraints:
 - $Z = K * Y$, with K a constant number;
 - $Z = X + Y$;

- $X \text{ lelin } Y$ ("less or equal" handled by the linear solver);
 - $X \text{ gtlin } Y$ ("greater than" handled by the linear solver);
 - other similar comparison operators ...
- General constraints like $\text{dif}(X, Y)$ which constrains X and Y to be distinct (disequation).

Meta-properties The second category of conditions contains "meta" constraints or properties.

- $\text{ground}(X)$ which is true if and only if the set of elements of the domain, which can be assigned to X and which satisfies the constraints, is a singleton.
- $\text{listnum}(L)$ which constrains L to be a list of numbers.
- $\text{listnum}(L, A, B)$ which constrains L solutions to be a list of numbers in the interval $[A, B]$.

3 Static Proof

The goal is to prove that a given program is correct, according to the assumed assertions for the predicates in this program.

3.1 Program Verification

The method used here, is to verify each rule independently, considering all the assertions of the program. That means a local analysis is used rather than a global one. The reasons are that it is simpler and more efficient, it matches an incremental compilation model very well. If for each rule, each call in the body fulfills the preconditions, and if the postconditions are implied, the program is correct. Therefore, for each rule of the program, the following is performed:

1. The current state of constraints is initialized to empty;
2. the preconditions of the corresponding predicate are added to the current state;
3. for each call which occurs in the body, if it is a constraint, it is added to the current state, otherwise:
 - the preconditions of the called predicate are checked (that is to say they are implied by the current state);
 - the postconditions of the called predicate are added to the current state (that is to say we assume the predicate definition is correct).
4. the postconditions of the rule are checked (to be implied).

Checking and adding conditions are performed by a meta-interpreter of assertions.

3.2 Checking Conditions

Depending on the class of the condition (predefined constraint or meta-property) a different procedure is used. For predefined constraints, a proof by inconsistency is performed, that is to say the opposite constraint is temporarily added to the current state. If the new state is not solvable, this means that the original condition is implied and the check is satisfied. Otherwise, a solution exists in the extended structure which means that the original condition is not implied by the current state and the proof fails.

For meta-properties the proof is done by an abstract interpreter in a classical way. In fact, both computations are done in a combined domain abstraction.

3.3 Abstractions

We do not give here a complete formalisation of the abstractions used. We just present a sketch of these abstractions. At present, each term is abstracted by two constraint systems (following the ideas described in [49]). The first abstract domain is used to study a meta-property (groundness). The second one is used to abstract the actual domain and the predefined constraints.

As an example a variable X will be abstracted by the list $[X_g, X_x]$ where:

- X_g is a boolean variable representing the groundness property;
- X_x is a representation of the predefined constraint state.

Let us consider now this program:

```
:- a(N,P) : (ground(N), cc(N,1,10)) => (ground(P), cc(P,10,100)).
:- b(N,P) : (true) => (P = 10 * N).
a(N,P) :-
    b(N,P).

...
```

In this example, we want to prove the assertion for the rule $a/2$. The prover abstracts the variable N by the structure $[N_g, N_x]$ and the variable P by the structure $[P_g, P_x]$. According to the method described above, the following is performed:

1. The current state of constraints is initialized to empty;
2. The preconditions of $a/2$ are added to the current state. Thus, the abstraction of the condition $ground(N)$ has the effect to affect the boolean variable N_g to 1, and the abstraction of the condition $cc(N, 1, 10)$ has the effect to constrain the variable N_x to belong to the interval $[1, 10]$;
3. Nothing is checked for the call to the predicate of the body ($b/2$), because there is no precondition for it. Therefore, the postcondition $P = 10 * N$ is added. The abstraction of the operator $'*'$ is defined as follows:

$$\frac{\text{times}K([Z_g, Z_x], [1, K], [Y_g, Y_x]) :-}{\begin{array}{l} Y_g = Z_g, \\ Z_x = K * Y_x. \end{array}}$$

So, the constraint $P_x = 10 * N_x$ is added to the current state.

4. The postconditions of $a/2$ are checked.

- Given the fact that $N_g = 1$ and that the constant 10 is ground, the equality of the rule $\text{times}K/3$ involves that P is ground ($P_g = 1$). The opposite condition of $\text{ground}/1$ consists in adding the abstraction $P_g = 0$. Obviously, this operation will fail.
- To check the condition $\text{cc}(P, 10, 100)$, the prover will try to add the opposite condition $\text{outcc}(P_x, 10, 100)$ to the current state. The result will be a failure of the interval solver.

So, we have checked the set of postconditions for $a/2$. Thus, the assertion of this rule is proved.

3.4 (In)completeness of the Solver

For the conditions which are predefined constraints, the completeness in the (real) intended domain is dependent on the solver involved. Let us consider the two Prolog IV solvers: the linear solver, which is complete, and the interval solver, which is incomplete. Depending on the solver used, a proof will succeed or not. As an example:

```
:- a(X,Y) : (X gelin Y) => (X gtlin Y).
a(X,Y) :- dif(X,Y).
```

In this example, the assertion of the rule $a/2$ can be proved. As a matter of fact, the constraints involved by the set of preconditions are first added to the current set of constraints. Thus, the constraint $X \text{ gelin } Y$ is added. When the body of the rule is handled, the constraint $\text{dif}(X, Y)$ is added. Then, to verify the postcondition, the goal is to check if the postcondition is implied by the current set of constraints. To do that, the opposite constraint (here the constraint $X \text{ lelin } Y$) is added to the current set, and a failure is expected. Given the completeness of the linear solver, this failure is obtained. So, the implication of the postcondition by the current set is proved.

Now, let us consider the example:

```
:- b(X,Y) : (X ge Y) => (X gt Y).
b(X,Y) :- dif(X,Y).
```

For the rule $b/2$, the behaviour is not the same. When the opposite constraint $X \text{ le } Y$ is added to the current set (which already contains the constraint

$\text{dif}(X, Y)$ added by the treatment of the body), the expected failure does not occur. So, the assertion is not proved.

A similar behaviour can appear when both solvers are requested in one assertion. Example:

```
:- d(X,Y) : (X gtlin 3) => (X gt 5).
d(X,Y) :- Y = X + 2.
```

If one wants to reduce the effects of the incompleteness of the interval solver, it is advisable in assertions:

- to limit non linear constraints to have a unique variable (and other constants) among their arguments;
- not to mix constraints handled by different solvers.

More generally, if we want to guarantee soundness of our system, we have to use only constraints handled by complete solvers. In a further work, we will try to specify exactly what conditions have to be filled by the constraints (in general and in our particular case with these two solvers) to avoid some basic failures of the proofs.

3.5 Full Example

Let us consider the famous program $SEND + MORE = MONEY$. This program can be entirely proved if we introduce assertions taking into account the considerations listed above. That gives the following source code:

```
:- solution(Send,More,Money) : (true) =>
    (ground(Send), cc(Send, 1000,9999),
     ground(More), cc(More, 1000,9999),
     ground(Money), cc(Money, 10000,99999)).
:- allintegers(L) : (true) => (ground(L), listnum(L)).
:- allDifferentDigits(L) : (listnum(L)) => (listnum(L,0,9)).
:- outsideOf(X,L) : (listnum(L) , real(X)) => (true).
:- difrat(X,Y) : (real(X), real(Y)) => (true).
:- enum(X) : (true) => (ground(X), real(X)).
```

```
allintegers([]) .
allintegers([E|R]) :- enum(E), allintegers(R).
```

```
allDifferentDigits([]) .
allDifferentDigits([X|S]) :-
    0 le X, X le 9,
    outsideOf(X, S),
    allDifferentDigits(S).
```

```

outsideOf(X, []) .
outsideOf(X, [Y|S]) :- difrat(X,Y), outsideOf(X, S).

solution(Send, More, Money) :-
    S ge 1 ,
    M ge 1 ,
    Send .+. More = Money ,
    Send = 1000 .*. S .+. 100 .*. E .+. 10 .*. N .+. D ,
    More = 1000 .*. M .+. 100 .*. O .+. 10 .*. R .+. E ,
    Money = 10000 .*. M .+. 1000 .*. O .+. 100 .*. N .+.
        10 .*. E .+. Y ,
    allDifferentDigits([M,S,O,E,N,R,D,Y]),
    allintegers([M,S,O,E,N,R,D,Y]) .

```

4 Run-Time Checking

When verifying a program, if a precondition cannot be proved for a given goal in the body of a rule, the goal is flagged as unchecked for this precondition. Then, the proof continues normally as if the proof of all the preconditions had succeeded. At compile-time, the code needed to verify the unchecked preconditions is inserted before the goal. The principle is the same for the postconditions. Most of the time, a run-time check is easier to perform than a static general proof because the constraints accumulated by the execution of the whole program specify arguments. Example:

```

:- sumlist(L,S) : (list(L)) => (ground(S), real(S)).

sumlist([],0).
sumlist([X | L],S1) :-
    S1 = S + X ,
    sumlist(L,S).

```

In this example which computes the sum of a list, no groundness is assumed by the preconditions, so for the second rule the groundness cannot be proved in the postconditions. The rules actually inserted are:

```

sumlist([],0).
sumlist([X | L],S1) :-
    S1 = S + X ,
    sumlist(L,S),
    run_time_check_post(sumlist([X | L],S1)).

```

For the same rule, if we replace its unique assertion by:

```
:- sumlist(L,S) : (ground(S)) => (ground(L)).
```

the groundness in the precondition cannot be proved for the goal *sumlist/2* called in the body of the second rule, so the rules actually inserted will be:

```
sumlist([],0).
sumlist([X | L],S1) :-
    S1 = S + X ,
    run_time_check_pre(sumlist(L,S)),
    sumlist(L,S).
```

where *run_time_check_post/1* and *run_time_check_pre/1* are procedures which retrieve the postconditions (or preconditions) of its predicate argument and check them.

5 Conclusion

We have presented an extension to constraint languages consisting in introducing assertions. This extension allows an automatic verification of programs, combining compile-time analysis and run-time verifications. The assertions can be expressed with a rich set of primitives, including constraints, and are very helpful for program debugging. The benefit is also to introduce a much better specification and readability of programs. Such an extension is currently introduced in the Prolog IV environment and seems to be a very valuable development tool. Now, it must be validated on complex real-life applications solving large scale problems.

References

1. Bergere, M., Ferrand, G., Le Berre, F., Malfon, B., Tessier, A.: La Programmation Logique avec Contraintes revisitée en Termes d'Arbre de Preuve et de Squelettes. Rapport de Recherche LIFO 96-06, Feb 1995.
2. Boye J., Maluszynski J.: Two aspects of Directional Types In Proc. Twelfth Int. Conf. on Logic Programming, pp. 747-763, The MIT Press, 1995.
3. Bronsart, F., Lakshman, T.K., Reddy, U.: A framework of directionality for proving termination of logic programs. In Proc. of JICSLP'92, pp. 321-335. The MIT Press, 1992.
4. Codognet, P., Filé, G.: Computations, Abstractions and Constraints in Logic Programs, International Conference on Computer Language, Oakland, 1992.
5. Deransart, P., Maluszynski, J.: A grammatical view of logic programming. The MIT Press, 1993.

6. Drabent, W., Maluszynski, J.: Induction assertion method for logic programs. *Theoretical Computer Science* 59, pp. 133-155, 1988.
7. Ferrand, G.: Error Diagnosis in Logic Programming. *Journal of Logic Programming* 4, 177-198, 1987.
8. Le Berre, F., Tessier, A.: Declarative Incorrectness Diagnosis of Constraint Logic Programs. *Rapport de Recherche LIFO 95-08 Université d'Orléans*, Feb 1995.
9. Vetillard, E.: Utilisation de déclarations en Programmation Logique avec Contraintes, PhD thesis, U. of Aix-Marseilles II, 1994.

A Return to Elegance: The Reapplication of Declarative Notation to Software Design

David A. Schmidt

Computing and Information Sciences Department, Kansas State University,
234 Nichols Hall, Manhattan, KS 66506 USA.*
`schmidt@cis.ksu.edu`

Abstract. Software design methodologies were introduced to manage the scale of complex systems built in imperative languages under demanding work conditions. To some degree, declarative notations have been ignored for systems building because they lack similar design methodologies. Methodologies useful to object-orientation, namely, software architectures, design patterns, reusable libraries, and programming frameworks, are proposed as a model to be imitated by the declarative programming community. The resulting “declarative design methodology” would hasten the reapplication of declarative notations to mainstream software design and implementation.

1 An Evolution of Programming

Certainly, there is no shortage of design methodologies for software, but at some point after the requirements descriptions, entity-relationship diagrams, and life-cycle models have been written, a programmer must convert descriptions into reality—into actual programs.

In recent years, this final stage—the writing of code—has been deemphasized; in particular, the insights of Dijkstra, Hoare, and Wirth regarding the refinement of specifications into correct, concrete code have been forgotten, and emphases have changed from “do it carefully” to “do it by deadline” and from “prove it correct” to “let the users and testing tools prove it wrong.” The contemporary argument is that software is inherently too complex for programmers to understand, let alone prove correct, and only brute-force automated tools can “understand,” that is, impassively locate errors. Regardless of its validity, the argument itself is dispiriting, for it implies that the human programmer can not understand what she is doing, let alone take satisfaction from knowing that she has completed the task correctly.

Why has the situation evolved so? There are certainly the pressures of the marketplace:

- The intensive demand for software, coupled with the competition to produce quickly new product to fill new niches, forces unhealthy deadlines onto programmers. It is as if a medical surgeon has been imposed with unrealistic

* Supported by NSF/DARPA CCR-9633388 and NASA NAG-2-1209.

quotas for completing surgeries with the justification that botched surgeries will be detected later by patients and repaired by user support staff.

- The high demand for programmers has forced a larger percentage of the general population into the profession, at a time when education in mathematics and the hard sciences in the (United States') secondary school systems is suffering. It is as if demand for surgeons motivated training more and more of the general public in performing "elementary" surgeries.

Programming is a difficult intellectual endeavor, and attempts to produce programs quickly or train large segments of the populace at programming are as likely to succeed as efforts to do surgeries quickly or train the general population as surgeons.

But there are also deeper philosophical questions about the evolution of computing:

- Thirty years ago, programming methodology was based on the need to conserve expensive computing resources: A programmer was forced to understand a specification, hand code a solution to the specification, and hand test or argue correctness of the coding before supplying the program to a computer for a one-per-day execution. The resulting program was an entity unto itself, to be described, analyzed, and even proved correct before it was supplied to a computer for a reading. In contrast, modern-day computing resources are cheap and make possible programming methodologies based on experimentation: A programmer understands a specification by trial-and-error, fashioning experiments/scenarios-in-code whose repeated executions lead to repeated repairs until acceptable output is obtained. Such a program is an entity that evolves within and depends upon the computer, and such a program attracts no interest as a literary or mathematical artifact.

Both methodologies have their merits and both methodologies can be abused, but the latter methodology must be accepted because it is here to stay.

- Unlike forty years ago, when programming was an esoteric activity, practiced by lonely specialists, it has evolved into an industrial activity that demands aspects of science, engineering, and architecture. The imprecise dividing lines between the scientific, engineering, and architectural aspects mean that a programmer specialized in one aspect of programming is asked to perform and even lead endeavors in the other aspects, often with disappointing results.

Indeed, the tension between the science, the engineering, and the architecture of computing threatens to tear the field itself apart into three subfields. If this happens, to which subfield will programming belong?

Given these pragmatic and philosophical issues, it is no surprise that elaborate software design methodologies have been developed to keep some control over the current, unstable state of affairs.

2 The State of Declarative Programming

No doubt, the use of imperative notations for doing the actual programming of a large system has accelerated the development of software design methodologies—when one writes programs in a notation whose fundamental unit is a primitive manipulation of program state, one is forced to use a separate design methodology to express program structure that is otherwise buried within the sheer scale of the program itself.

In contrast, one sees no similar intensive development of software engineering methodologies in the declarative programming world, possibly because declarative notations do not perform low-level manipulation of program state—the crisis of program scale does not arise as quickly. Also, because it is state-free, declarative notation serves well as both a design notation as well as a programming notation. Hence, the design and programming processes merge, almost invisibly.

Such elegances should be advantageous, but they have a surprising drawback: Not enough attention has been paid to the architectural and engineering aspects of declarative programming. In this regard declarative programming is immature, evolving little from its original conceptions. If one was to list the major advances in declarative programming within the past twenty years, one might list specific language constructs such as arithmetical-and-subset constraints, pattern matching, and bounded subtyping rather than architectural or engineering advances such as “declarative entity-relationship diagrams,” “declarative design patterns,” and “declarative programming frameworks.”

3 What Can Be Learned from Object-Orientation?

Perhaps the most interesting aspect of object-orientation is its attempt to address problems of scale and reuse within a universe of standardized, connectable components. In its ideal manifestation, object-orientation makes a programmer do most of her programming during the design stage, when an architecture is selected, components are precisely specified to instantiate the architecture, and connection points between the components are explicitly defined and connected. The final stage of implementation is reduced to the selection of off-the-shelf components that meet specification (or the coding of new components that provably meet specification).

Of course, the reality of object-oriented programming rarely matches this ideal, but there is much to admire, most notably the attempt to integrate architecture, engineering, and science in the programming process. To reduce this to a (slightly awkward) slogan, *using object-orientation, one programs by engineering scientifically-well-founded objects to fit into an architectural layout*. A pleasing consequence is that the Dijkstra-Hoare-Wirth approach to method specification and correctness is compatible with and even integral to the object-oriented programming process—*components must meet specifications*.

Important natural consequences of object-orientation have been the establishment of standard libraries of components, the documentation of “design pat-

terns” of object combinations that let one assemble systems that match standard “software architectures,” and the production of “frameworks,” which are partially completed software architectures that can be customized for specific applications. None of these developments are language-based, nor are they necessarily paradigm-based—indeed, they take on an identity of their own.

4 A Possible Future for Declarative Programming

The elegance of declarative programming has never been called into question, but its applicability to real-life scenarios has. Some common complaints are

1. inefficient implementation, e.g., heap-storage management
2. lack of well-documented support libraries for common applications, e.g., graphical interfaces, networking, and file manipulation
3. lack of a “killer app,” that is, an important application area that is ideally suited to computing by declarative means
4. a programming style that is too difficult for the average programmer to master

Some of these complaints can be refuted—consider Java’s success as a refutation of Complaint 1—and others appear to have strong validity but are slowly being addressed (e.g., Complaint 2). Perhaps others are a matter of fate (e.g., Complaint 3), and others are speculation (Claim 4). But a new complaint must be added to the list:

5. lack of a mature software design methodology

It is time for the declarative programming community to adopt a design methodology that mimics the best aspects of existing methodologies, in particular, those aspects of object-orientation that emphasize integration of architecture, engineering, and science in the programming process. Such a methodology should encourage the description of standard declarative software architectures, the collection of useful design patterns, and the formulation and documentation of standard libraries for important application areas.

The benefits from these efforts will be at least two-fold:

- The efforts will foster the maturity of the declarative programming field.
- The efforts will establish a vocabulary that is understandable by the many in the mainstream software engineering community, easing adoption and integration of declarative programming into the mainstream.

Because of its elegance, declarative notation can serve as the language of specification (of architectures, of design patterns) as well as the language of implementation.¹ Such an elegance has a good chance at impressing practitioners of software design. And at the very least, those who already believe that

¹ But one should never underestimate the impact that a visual notation brings. Visual representations of declarative constructs become especially useful at the design level, and their development should be strongly encouraged.

declarative notations are merely “specificational” in nature can use the notations to devise specifications.

Regardless of the impact of these efforts on the software engineering mainstream, it is time for declarative programming technology to take the step towards codifying, documenting, and demonstrating its design techniques as a comprehensive “declarative design methodology.”

5 Conclusion

It is discouraging that modern-day systems building is so complex as to make it difficult or impossible for any one person to understand, appreciate, and take pride in the completion of a program. To make systems building manageable and understandable by the individual, an elegant integration of the architectural, engineering, and scientific aspects of programming must be undertaken. Declarative programming methodology has much to offer in this regard, and the time is right for the reapplication of declarative notation, in the guise of a declarative design methodology, to the software development process.

6 References

- Timothy Budd, *Understanding Object-Oriented Programming With Java*, Addison-Wesley, 1998.
- Erich Gamma, et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
- Mary Shaw and David Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996. Or, David Garlan and Mary Shaw, An Introduction to Software Architecture. In V. Ambriola and G. Tortora, eds., *Advances in Software Engineering and Knowledge Engineering*, World Scientific Publishing Company, Singapore, pp 1-39, 1993. Also available as http://www.cs.cmu.edu/afs/cs/project/able/www/paper_abstracts/intro_softarch.html.

ECLiPSe : Declarative Specification and Scaleable Implementation

Abstract

Mark Wallace and Joachim Schimpf

IC-Parc,

William Penney Laboratory, Imperial College, LONDON SW7 2AZ.

{mgw, js10}@icparc.ic.ac.uk

Some years ago the vision of computer scientists was to enable end users to specify *what* they wanted, and leave it up to the computer *how* to work it out. Two important issues concerned the specification language in which end users could express their requirements, and the transformation of the specification into runnable code.

Kowalski, recognising that the specification must be an integral part of the final system, proposed that efficient algorithms be built by simply adding control to the original formal specification. This thesis is summed up in his maxim *Algorithm = Logic + Control*. Constraint Programming, and a fortiori Constraint Logic Programming (CLP), is based on this equation.

By ensuring that the problem specification is embedded in the final program, constraint technology directly supports the development of correct algorithms solving the problem which the end user needs to be solved. Changes in the end user requirements are added to the formal specification because that is the shortest/only way to add them into the code!

The "classical" constraint program is a nice embodiment of Kowalski's equation. Its form is:

```
solve(ProblemVariables) :-  
    imposeConstraints(ProblemVariables),  
    labelling(ProblemVariables).
```

A solution to the problem is an assignment to the problem variables which satisfies the constraints. The constraints embody the problem specification: they are a logical expression of what is required of any solution to the problem. The labelling procedure is the code that explores the solution space searching for solutions. This procedure embodies the control which is necessary to enable the problem to be solved with reasonable efficiency.

However the classical constraint program suffers from severe limitations. One problem is that too much behaviour is built-in. Constraint behaviour is built-in, and can only be changed by writing the constraint in a different way: this violates the separation of logic and control. Search behaviour is also built-in: usually depth-first search (though with a dynamically constructed search tree).

More importantly, alternative approaches such as mixed integer programming and stochastic search are not usually accommodated within the classical con-

straint program form. Clearly these techniques should not compete but should be harnessed together.

The CLP platform ECLiPSe, is designed to support two goals:

1. Relating problem specification to problem solvers
2. Relating problem solvers to each other

Firstly the ECLiPSe language separates logic from control, and allows the control to be expressed in a high-level syntax that minimises the code that needs to be written to achieve the required hybrid algorithm. Only with such a high level syntax is it possible for the programmer to experiment with a reasonable variety of algorithms and find a good one.

Secondly a way of building algorithms in ECLiPSe has been developed that allows the different building blocks to be mixed and matched. The application developer can combine in one program, for example, linear programming, interval propagation and local search.

ECLiPSe is used for solving difficult Large Scale Combinatorial Optimisation (LSCO) industrial problems in the areas of planning, scheduling and resource allocation. ECLiPSe applications have been delivered to the transportation industry, the supply chain industry, and the network services business.

Author Index

- E. Åström 335
- A.K. Bansal 275
R. Bianchini 122
P. Bork 275
M. Bozzano 46
- M. Cadoli 16
M. Carlsson 335
- L. Damas 243
G. Delzanno 46
B. Demoen 106
S.W. Dietrich 164
T. Dowd 211
I.C. Dutra 122
- C. Elliott 61, 91
- M. Ferriera 243
- H-J. Goltz 320
- R. Hakli 179
J. Hook 196
P. Hudak 91
- D. Jeffery 211
- W. Kahl 76
P. Kreuger 335
- C. Lai 350
D. Leijen 196
P. Letelier 31
- M. Martelli 46
V. Mascardi 46
H. Mattsson 152
D. Matzke 320
E. Meijer 196
G.E. Moss 1
- I. Niemelä 305
H. Nilsson 152
M. Nykänen 179
- L. Palopoli 16
J. Peterson 91
- I. Ramos 31
R. Rocha 137
C. Runciman 1
- K. Sagonas 106
P. Sánchez 31
V. Santos Costa 122, 137
A. Schaerf 16
J. Schimpf 365
D.A. Schmidt 360
B. Siddabathuni 164
F. Silva 137
M.G. Silva 122
T. Soininen 305
Z. Somogyi 211
- H. Tamm 179
- E. Ukkonen 179
S.D. Urban 164
- W.W. Vasconcelos 259
D. Vasile 16
- M. Wallace 365
R. Watson 290
C. Wikström 152
- H. Xi 228
- F. Zini 46